A tree-structured system.

This report is a summary of the discussions we (i.e. Bomhoff, Bron, Feijen and I) held during January 1970, in a first effort to give some shape to the Programming Laboratory Project. It is a highly condensed summary: many detours and blind alleys will be left unmentioned. Besides that, it was the first month that this group of four people tried to think together and many a day was just spent in getting used to the other's way of coining words, of twisting sentences and of struggling with half-baked ideas.

We have assumed that programs are conveniently conceived and considered as "necklaces", i.e. as linearly ordered sets of "pearls". (I know only too well that the only convincing examples I have for this assumption are rather small examples, as trying it out for larger examples is too expensive now. I also dare to be content with that experimental evidence, at least for the time being, for after all, I arrived at the concept of pearls in a necklace by rather independent means and the examples worked out were only confirmations.) The relatively small examples gave already rise to rather long necklaces (6 pearls in a small program), it also became clear that my main goals (provability, adaptability) are served by small pearls. Therefore, if a large program can be made in a way in which these main goals are reasonably well achieved, we must be prepared to process programs composed as very long necklaces. Our first effort was to get a feeling for the implementational consequences of the great length of the necklaces.

The pearls are ordered in a linear hierarchy, at the bottom side we have "the machine", at the top we have -if we go to extremes- the famous pearl "Do All Work", the function of each pearl being to rebuild the machine underneath it into a more attractive machine to be used above it.

In the beginning it turned out to be very hard to come to grips with our problem, to discover what we were supposed to be talking about. The sky became somehwat clearer when we discovered the main source of our difficulty. Assuming that one has at one's disposal a sufficiently automated tool to compose programs from pearls, then we believe to have a tool that caters reasonably well for operational and representational abstraction as far as the subject matter of these programs is concerned! But the problem we were tackling was "how to represent pearl texts inside the machine so that they can be interpreted reasonably efficiently". It is a kind of bootstrapping problem, but it took us some time to discover that this was the case. (The dilemma is perfectly clear if we think of subroutines, i.e. the well-known form for operational abstraction: whatever new operational primitive you wish to introduce, you can make a subroutine for it. But you cannot apply this standard technique when the primitive you want to make is.... the subroutine jump!)

As announced in EWD279, one of our first concerns would be a binding policy. To rephrase the question: when you go along the necklace from bottom to top, you will find existing concepts losing their applicability and new concepts becoming applicable as you go along. In the pearls at the different levels, the object code must refer to them and the question becomes: in what terminology?

We have played with the following idea. Each pearl "owns" the concepts it introduces and distinguishes between them via a local index. Besides this, it distinguishes via an "underindex" between the concepts that are already there. Furthermore, we assumed that each concept is suitably identified globally by a couple, consisting of a global identification of the pearl owning it and the local index used by this pearl to distinguish between its property.

Such a convention implies for each pearl a kind of "binding list" for the interpretation of underindex values and immediately one is faced with the question: what entries do we find in this binding list? In one extreme the entry contains by definition the bit pattern by which the immediately underlying pearl identifies the concept. This has the unquestionable advantage that whatever is built on top of a pearl is, in its representation, independent of the necklace structure underneath the pearl in question. Attractive as this might be, it seems to lead to unacceptable consequences:

1) the individual binding lists tend to get very long, for the underindex values for each pearl have not only to distinguish between the existing concepts it really refers to, but also between those, that it only transmits to above.

2) the accessing mechanism presents itself as a kind of indirect addressing in which the length of indirection is equal to the distance between the referring pearl and the owning one. This latter aspect makes this solution hardly acceptable for a system that would like to promote programming in terms of small pearls and therefore of long chains! With N layers the accessing speed would be proportional to 1/N and this we have rejected as general policy.

So a shortcut seems indicated and we have considered the other extreme: in the binding lists (or directly in the pearl object texts -this is only a minor difference) the accessing mechanism finds a bitpattern which directly identifies the concept in a global terminology, i.e. the access time will be independent of things such as pearl distance and chain length. In other words,we introduce a global terminology and allow this this global terminology to diffuse through the binding lists or pearl texts. (Intermediate forms are conceivable, where at certain "major cuts" in the necklace a completely new terminology is introduced for all or part of the concepts transmitted to above; we shall not pursue that now.)

This suggests a binding policy in the direction from bottom to top, or, if the necklace is generalized into a tree, starting at the root. Consider the stage where the tree has grown to a certain level and a next layer has to be added. The binding function of the translator then has the following function. The source text has to be interpreted in terms of the concepts, relevant at that level, a "source context". We can think of the source context as being defined along the necklace and by searching from top to bottom each source concept will be found and can be translated into the "object context" which covers all concepts along the chain. (In ALGOL 60, the only way to make an identifier, declared in an outer block, inaccessible in an inner block, is "redeclaration" of the same identifier in the inner block. A pearl could contain, besides the concepts it introduces, an "introduction list", say, also a "removal list", i.e. a list of existing concepts no longer applicable in the structures on top of it.)

We then see a mechanism emerging in which the object context only grows and grows as new pearls are added, while the binding function of the translator will have a strong protective function, i.e. it has to see to it, that object text -although it has the potential richness-never refers to a concept outside the corresponding source context. It is this mapping of source context into object context which strikes us now as the main aspect of the binding function: for lack of associative memories it is a painful process and for that very reason it seems a suitable candidate for what is usually called "the translation fase". It implies that the bottom part can be bound regardless of what will be built on top of it, it also implies that (more than trivial) rebuilding of a bottom part calls for 'rebinding" of what has been built on top of it.

At present I expect us to take this decision: it is the only way I can see such that the performance does not degrade too much with increasing chain length. This decision in favour of "natural order of growth" will only be taken after much hesitation -as a matter of fact we already did so. The reason for hesitating was that taking this decision struck us as giving up hope "to change a program while it is running". It is quite clear that this will be difficult (if not impossible!) and for reasons of design strategy one must impose an upper limit on the period of time during which one allows oneself to be paralyzed by the (still apparent) anattainability of such a goal. I think it has paralyzed us long enough. The kind of flexibility necessary (but insufficient) for that goal seems very expensive; at present I hope that the decision of "natural order of growth" provides a framework in which the problem of dynamic program changing permits a more precise formulation.

* * *

The next thing we talked about -although not very extensively- was the representation of the object code, in particular the way in which it will be parsed under control of its order counter. If working in an interpretative mode slows down the performance by a factor of R, it is absolutely inacceptable if we have to pay that factor for every new pearl: for a necklace of length N we would then get a slowing down by a factor R↑N.

Our conclusion was very similar to the one we drew in connection with identification, where we concluded that the identifying bit patterns in the object code should control rather directly a global accessing mechanism, independent of pearl height etc. We concluded that at all levels the instruction stream should be parsed by essentially the same "instruction cycle", a requirement which asks for a very flexible instruction format.

Our current guess is to combine the format flexibility of stack machines which process the instruction stream as a string of syllables with the semantic flexibility of the "extra codes" as have been implemented in one or more English machines. To be more precise: instruction execution takes place with the aid of a stack, the top section of which is considered as containing the active registers, instruction register included. The instruction cycle reads syllables from the program text that are essentially copied on top of the stack. They are subjected to a minimum amount of interpretation: a distinction is made between "passive" and"active" syllables (say: on account of a dedicated bit in the syllable representation); as soon as an active syllable has been copied on top of the stack an operation is started as described (primarily) by the contents of the top of the stack. This should cater for the format flexibility. We hope to achieve the desired semantic flexibility by requiring that at each level active syllables can be used irrespective of the question whether they are conceived as identifying "a built-in function" or "a primitive explained by software". (This is essentially the notion of "extra codes"). If we succeed in doing this, we may build a software system in which the top layers are -structurally, at least- unsensitive to the question how many of the lower layers have been absorbed by the hardware. (This goal comes very naturally. At top side we wish the boundary between system building and system usage to become more vague; similarly at the bottom side with the (conceptual) boudary between hardware and software.)

We did not work this out in greater detail, we did discuss the question of "ideal syllable length". Our first (and second!) impression was that on the one hand it will be very hard to defend a specific choice for syllable length, but that on the other hand (on account of the global activity of the instruction stream parsing) it will be still harder to avoid the decision. Dear old Sartre!

* * *

A next discussion, again, took a long time to get to the point. We saw a tree-like structure emerging and the point in question turned out to be "for what purposes do we hope to exploit this tree-like structure?".

Our original, first candidate was the notion of "parallel and nested conceptual universes". Let me describe how this notion came into being. In the (constant) part of the THE-system we have a single set of layers: at one level we still have a drum, on top of it we have a new universe, in which we have a virtual store, but no drum anymore, the drum being "used up" for the implementation of the virtual store. But here one could think of a "split level", on the one hand a universe in which there is virtual store and on the other hand one in which (part of) the core store and (part of) the drum are still as available as ever,i.e. we have made two worlds, one in which the concept of the virtual store is applicable and one where it is not. From then onwards, two mutually independent subsystems can be developed. (In the top layer of the THE-system we actually do have the split level: the five PM's, which, when loaded with programs, do create five mutually independent interpretative mechanisms, viz. the interpretative mechanisms for the five sets of user data.) The conceptual mutual independence of parallel branches is considered to imply potential parallellism of the sequential processes related to them: a single sequential process could not be related to two parallel branches, more than one sequential process in one branch was not excluded a priori. For after all: the function of a layer is "to rebuild a machine", the given machine as well as the target machine may have mutually asynchronous components. We shall return to this question in a moment.

A second purpose for which the tree-structure can be exploited is a straightforward one. If parallel branches are conceptually independent, the going on of a sequential process somewhere high up in the tree can only depend (primarily) on concepts explained along the path leading from its position in the tree downwards to the root. This has a number of consequences.

1) For the effective executability of a sequential process, various degrees of "presence of information" can be defined. The highest form of presence is presence in primary store, the second form is dumped on secondary store but with the descriptors itself still present in primary store, the third form could have even the descriptor absent from primary store, etc. By placing the descriptors themselves in the appropriate position in the tree, it is conceivable that "off-path" descriptors could be dumped from primary store without introducing the well-known horror of the "recursive page fault".

2) As primary reference will only be along the path leading downwards to the root, we only need an identifying terminology distinguishing elements <u>along that path</u>. As shown (by Brian Randell and myself) this circumstance is strongly suggestive for various speeding-up devices -e.g. the "stack display". Besides that, it makes parallel branches truly independent in the sense that in the representation of their texts the same identifying terminology can (and in all probability will) be issued by the translator. (Cf. numbering the Dutch babies for the sake of identification in order of moment of birth: such a convention requires synchronization all over the country between all maternity wards!)

We now return to the relation between the tree and sequencing. Consider a single chain of pearls. In one of the higher pearls a primitive is invoked. On the level where it is used, such a primitive represents a single action, and it is only when we show a more microscopic interest in what is happening that the activity of such a primitive presents itself, in its turn, as the execution of

a sequential program. With this in mind we turn our attention to the critical sections as implemented in the THE-system. What are they? They are "single actions" in the program in which they occur. It is only because the basic machinery allows for a finer grained parallellism that we must encapsule the sequential sub-process by means ofa P-V bracket pair operating on a semaphore introduced for mutual exclusion. It is the kind of mutual exclusion that on a lower level -say the cross bar switch- is guaranteed by the hardware. This observation is strongly suggestive for the following approach.

If a number of parallel processes contain sections critical with respect to each other (THE terminology), we can "take them out", consider them on the level of the parallel processes as primitives and refine them in the common trunk, in the implicit understanding that the common trunk will serve "one at a time", in exactly the same way as in multiprogramming, where the single central processor switches from one program to another only between instructions.

Arguments in favour of exploiting the tree structure for mutual exclusion as well, are the following ones. (To a certain extent this list amounts to an enumeration of anomalies in the THE-system!)

1) Critical sections have been introduced for the unambiguous inspection and setting of common state variables; in the design of the THE-system it became a wise discipline never to access these common variables outside critical sections, not even in those cases where a piece of (usually very tricky) reasoning could justify it. In those years, the wisdom of this discipline was a "scientific discovery"; now we have made it, we must conclude that the common state variables should not be accessible from outside critical sections. As long as we regard them as normally accessible -but programs are not allowed to access them outside their critical sections- this calls for specific protection measures. By moving the critical sections to a common "secretary", down in the trunk, this problem is solved by structure.

2) In the THE-system we use the same P- and V-operations on two very different kinds of semaphores. This is expensive and (worse!) confusing. I remember our "scientific discovery" that these two classes of semaphores were both attractive and sufficient. Again, this difference should preferable by represented by structure.

3) For reasons of reaction time we had to introduce for the PM's a priority (as far as processor allocation was concerned) that varied in time: as soon as as PM entered a critical section, it got the maximum priority as long as it stayed in the critical section: not the process itself was in a hurry, but its "being in a critical section" caused the urgency. In the necklace model, activity in the higher pearls only takes place by virtue of the activity of lower pearls and in that model it is quite natural to assign the highest priority to the lowest pearls. Also the priority rule of the THE system as far as critical sections are concerned, finds a natural place in the hierarchy we are considering.

4) Hendriks has already pointed out that if our only mutual synchronization is in the form of mutual exclusion via perhaps many mutual exclusion semaphores, deadly embraces are guaranteed to be absent, provided the bracket pairs are nested and the semaphores associated with the nested pairs are ordered in a tree fashion, i.e. exactly the kind of ordering we are considering! (In the THE-system his observation has not played a very important role because we only had two mutual exclusion semaphores. Having only one processor there was not much economic pressure to introduce more of them. But his remark was perfectly valid.)