# A Model for and Discussion of

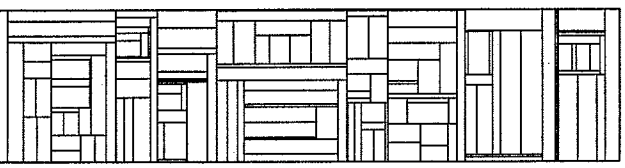
# MULTI-INTERPRETER SYSTEMS

by

Michael J. Manthey

DAIMI PB-73
April 1977

Institute of Mathematics University of Aarhus
DEPARTMENT OF COMPUTER SCIENCE
Ny Munkegade - 8000 Aarhus C - Denmark
Phone 06-12 83 55



# A Model for and Discussion of Multi-Interpreter Systems

by

Michael J. Manthey

## Abstract

By a multi-interpreter system is meant a system in which programs execute by virtue of being interpreted by other programs, which themselves may either be interpreted (i.e. nested interpreters) or run directly on the host machine. The model reveals the anatomy of interpreters and how these differ from procedures, and exhibits links to protection domains and multi-processor architectures.

# A Model for and Discussion of Multi-Interpreter Systems

## 1. INTRODUCTION

The purpose of this paper is to present a model for systems containing multiple interpreted machines. Subsidiary to this goal are the indications provided of how the model can provide a framework for the implementation of such systems.

It is not difficult to understand how interpretion as a general implementation technique has come to lie in relative disuse after its brief flowering in the mid-fifties:

(i) interpreters were slow (even by that day's standards);

(ii) they are still slow (e. g. LISP, SNOBOL, APL);

(iii) they introduce yet another 'machine' for which primary (e. g. compilers) and support software must be constructed and maintained;

(iv) they outlived their usefulness as the most baroque features (which interpreters were designed to hide) of the early machines were eliminated through the successive hardware generations.

If, however, we reexamine these truisms in the light of contemporary developments, both hardware and software, we find the following, respectively:

(i) Interpreters can be slow, but don't have to be. This is demonstrated by the fact that most computers are now implemented via micro-programs i. e. interpreters.

(ii) To be sure, LISP, SNOBOL, and APL, to mention the best known examples, are slow, but compared to what? The fact of the matter is that it is now recognized that the speed with

which a program produces 'answers' is dependent not on execution speed alone, but also the human time required to formulate the problem in machine-solvable form. The popularity of the above mentioned interpretive languages is without doubt due to users' recognition of how they can best invest their time.

(iii) The fact that each new interpreter (i.e. "machine") which is introduced into a system requires its own support software is indeed a problem [8]. If a way is not found to 'connect' the new machine with the existing one, then editors, file system, operating system etc. must be constructed for the new machine, which then executes oblivious to the host system. [This is effectively what was done for 1401 emulation on the 360's, except that all the support software for 1401's already existed.] On the other hand, if one could ensure that new machines could avail themselves freely of the host system's other software (which in general executes on other machines), then the problem yields to two subproblems:

a. presuming the new machines are primarily 'language' machines i.e. designed specifically to be a good host for some particular language, is it possible to produce the many compilers that will be needed for these machines,

b. is it possible to produce a general linkage which will allow invocation of software which runs on other machines?

Given contemporary compiler-generation technology and the fact that producing code for a custom-tailored language machine is extremely straightforward, (a) does not present insurmountable problems. The solution to subproblem (b) is presented later.

(iv) Thus it is incorrect to infer that interpreters have outlived their usefulness. [Indeed, today's hardware can be considered to be just as baroque, by today's standards and program - development difficulties, as the early machines by theirs.]Further evidence of this continued usefulness can be found in (1) the reluctance of many manufacturers who are developing multi-interpreter systems to discuss the techniques involved, and (2) the implications of widespread adoption of general interpreter-based software,

namely a sudden fluidity in the software marketplace as users discover that it is possible to change mainframes without leaving masses of expensive software behind them. This is an argument for higher-level languages and not interpreters, but language-machines (as opposed to much other system software) seem to ease the portability problem.

The discussion to this point has been based on what could be called every-man's knowledge and familiarity of interpreters. If we now assume the existence of a computer system based on the harmonious coexistence of multiple interpreters, then the following phenomena, to be discussed in greater detail in the body of this paper, are discernible:

1. Generalized interpreter hiararchies (i.e. nests of interpreters) require the retention of activation records beyond the point in time when execution leaves their context. Hand in glove with this is the striking resemblance between the generalized cross-interpreter procedure invocation and the concept of 'domain change' [21, 22, 23] in the context of filing and protection systems. It is therefore natural to equate activation records with segments, thus removing retention management and garbage collection to the realm of the file system, whose job is already and precisely retention and garbage collection.

2. The nature of the generalized cross-interpreter call is such that automatic parallel processing at the procedure level is a very natural extension. There is furthermore no constraint on these multiple physical processors that they be functionally identical. Thus there is a good possibility for building structured systems which exploit the low price of LSI micro-processors.

3. A layered approach to program development can in some cases be better realized through interpretative hierarchies than through procedure hierarchies, since access to primitives at the next lower level is both supplied and limited by the set of 'instructions' which each interpreter in the hierarchy is prepared to execute. Thus protection against invalid calls is independent of the lexical structure of the global context (as practiced by Burroughs in the B6700 [5]);

the built-in hardware protection, and the file system, not to mention project-programming standards. The resulting protection resembles, more than anything else, the result of building a system out of Simula classes e.g. Hoare [4].

4. The admission of retention as the standard discipline encourages both data and the applicable procedures being gathered together in that natural grouping, the activation record. This in turn leads to the accessing of data being viewed exclusively as the activation of access procedures. From the points of view of both protection and delayed binding, such would be a desirable development.

## 2. INTERPRETER NESTS

In this section, we derive some of the characteristics of interpreters. In order to ensure that these characteristics are universally applicable to all interpreters, we consider the general case of a series of interpreters nested within each other. Of course, as is well known from actual practice, deep nests of interpreters can be impractically slow, but there are nevertheless situations where modestly deep nests (depth = two or three) can be useful. [Examples of several such situations are presented in section 4.2.]

We begin with an intuitive example: consider a program FRED which runs on an X-machine which runs on a Y-machine which runs the H-machine, the hardware machine. In other words, FRED is formulated in X-code instructions, the X-machine program is formulated in Y-code instructions, and the Y-machine program is formulated in terms of the hardware H-machine's instructions. Assume now, for the sake of simplicity, that X-code, Y-code, and H-code all consist of only two instructions, Fetchinstruction and Executeinstruction, though the effect of each pair is presumably different for each machine.

Fred's program counter (PC) points to an instruction $I_1$ of algorithm FRED, the interpretation of which involves the execution of a Fetch ($F_x$) followed by an Execute ($E_x$) by the X-machine. In like fashion, $F_x$ requires the exe-

cution of the instructions $F_y$ and $E_y$, which in turn require the execution of $F_H$ and $E_H$. $F_H$ and $E_H$ are in fact the only directly executable instructions which exist in our example, and therefore represent the only instructions with which execution can begin. Thus by assuming that $PC_H$ points to an $F_H$ we get Figure 1a. After execution of this first $F_H$, $PC_H$ points to $E_H$, the execution of which completes the (interpretive) execution of $F_y$, which in turn implies the start of execution of $F_x$ (Figure 1b). Figures 1c and 1d show other steps on the way to the execution of $I_1$ in the program FRED. Finally, as shown in Figure 1e, we are on the verge of having completely executed $I_1$: execution of the final $E_H$ completes execution of the final $E_y$, which completes execution of the final $E_x$, which completes execution of $I_1$.

This example demonstrates two points:

1. The speed of execution of the instructions of FRED is exponentially proportional to the depth of the nest. The only (trivial) exceptions to this law occur when each instruction at a given level is fully executed (including PC maintenance) by a single instruction at the next lower level.

2. Control in a nest emanates from the bottom upwards, and not, as would seem more 'natural', from the top downwards. To put it another way, the control relationship between e.g. $I_1$ and $F_x$ is not that of a procedure call on $F_x$, but something else entirely. This subtle but important point finds several practical applications as will be seen later. Furthermore the example as it stands can be viewed as the 'dead start' of the nest, which as pointed out earlier can only begin with the direct execution of H-machine instructions. See Figure 2.

## 2.1 The Anatomy of the Hierarchy

It should be immediately clear from the example that the existence of multiple levels of interpretation carries with it a division of both function and access. In the case of function, each level (i.e. each interpreter) has available those instructions which the interpreter beneath it is willing to execute, and only those. Thus a wall, protective of function, is erected between reach level.

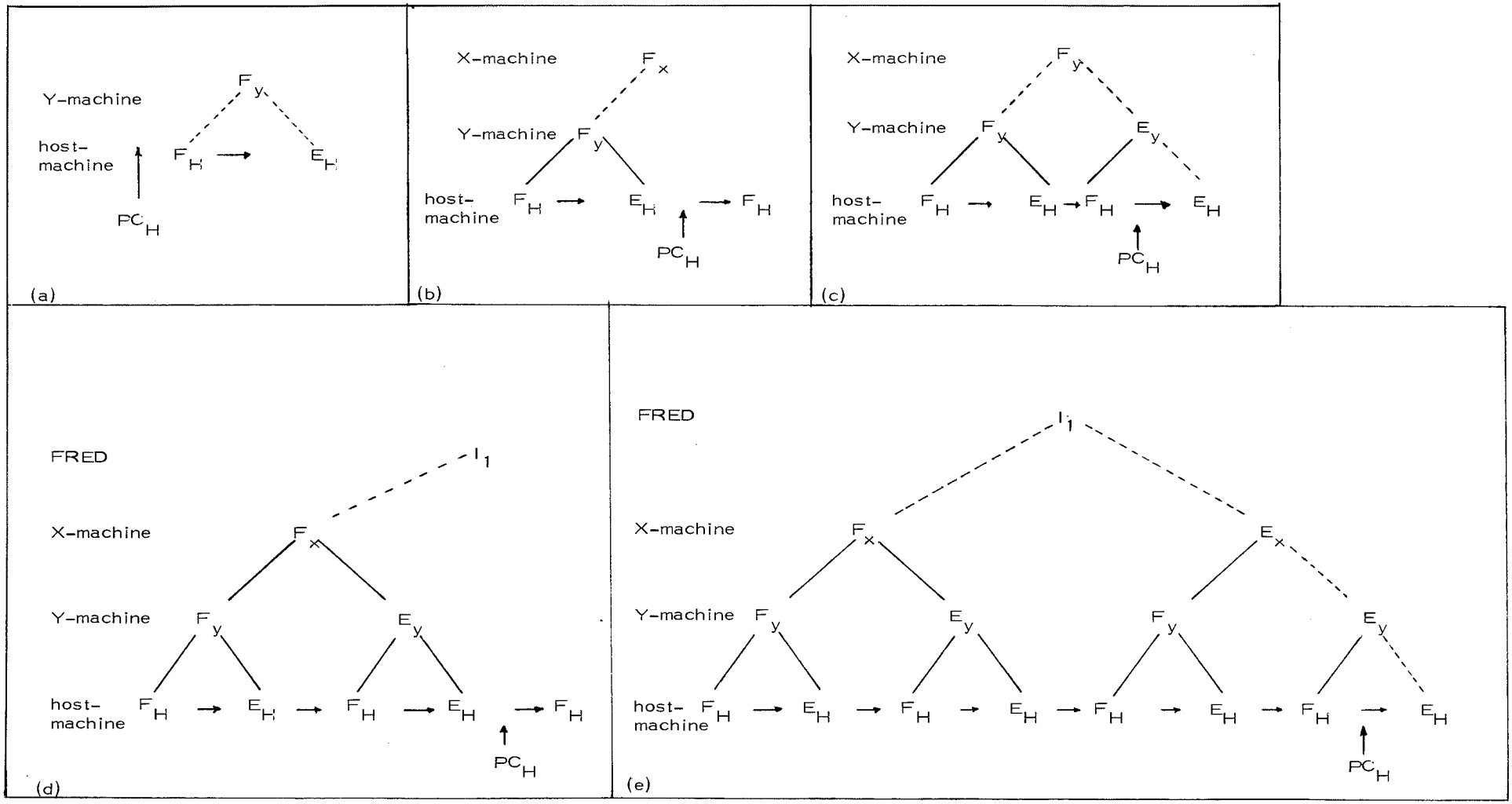



Fig. 1. Execution in an Interpreter Nest - the solid lines imply completed execution of instructions, the dotted lines, 'tentative' execution. [$PC_H$ points to the left of the next instruction to be executed.]

Fig. 2. Real-World Analogy to the Bottom-Up Control Regime

- The juggler at the top of the 'pyramid', being the only one able to move his hands, corresponds to the executing algorithm. Bottom-up control is illustrated by the fact that only the bottom juggler (corresponding to the host machine) can walk (execute). The analogy succeeds also for Cross-Interpreter Return but not for Call (unless a skyhook is used to set up the pyramid). [See Section 3.]

In the case of access, the situation is considerably murkier, and is treated in the following paragraphs.

A program can be considered as a collection of activation records [7] representing the environment in which the program and its currently active procedures operate. Johnston [6] expresses the data access environment of a given activation record as being reducible to a single value called the environment pointer (ep). In like manner, the precise point of execution within the algorithm is abstracted to a single value, the instruction pointer (ip). ip is commonly viewed as equivalent to PC, but it also includes ephemeral states and residual control [26] in the controlling machine. The pair (ep, ip) therefore represents the entire state of a given procedure activation. At any given point in time, there are an arbitrary number of (ep, ip) pairs in existence, one representing the current point of execution in the algorithm, and the others representing previously active but currently passive procedure activations. We express the totality of these pairs at a given point in time as (EP, IP). (It is important to distinguish between EP, which is a constantly changing collection of ep's, and an ep, which represents a collection of variables which once allocated remains essentially static in its composition.)

Given that an interpreter is a program, it can be represented at some point in time as (EP, IP), and in particular its total data environment as EP. We now return our attention to the example and ask "where is Fred's PC?" i.e. in which of $EP_{Fred}$, $EP_X$, $EP_Y$, and $EP_H$ can $PC_{Fred}$ be found? In a very general sense, it might be in any or all of the listed environments, and perhaps more than one simultaneously. However, in a more structured system there are really only two likely candidates: $EP_{Fred}$ and $EP_X$. We find it most fruitful to consider $PC_{Fred} \in EP_X$, and if in fact $PC_{Fred} \in EP_{Fred}$ as well, then this latter is a copy of that maintained by the underlying machine, in this case the X-machine †. It is easy to find existing hardware machines which demonstrate all possible combinations of this e.g. the PC exists solely as a (user) program register, that it is totally invisible to the user program, etc. Such variations have been introduced by designers to achieve certain hardware and/or architectural design goals, and each variation has its advantages. However, what

---

† We have restricted this discussion to the PC. A machine's STATUS when present, is viewed analogously and therefore will not be discussed further.

we are interested in determining is a useful canonical form for an interpreter, and from there to determine the global (i.e. system-wide) effects of same.

Having now isolated the PC at any one level to the EP of the immediately underlying level, we are now in a position to state a great deal more about interpreters and their nests. For some arbitrary program P on machine M we have

$$P: [\underline{ep}_P, \underline{ip}_M]_t$$

where ep instead of EP (etc.) are specified since we are interested in the current point of execution at time t.

Consider as an example a program BOB which runs on a hardware machine H:

$$BOB: [\underline{ep}_{Bob}, \underline{ip}_H]_{t_0}$$

If BOB invokes a procedure Q we have

$$BOB{:}Q: [\underline{ep}_q, \underline{ip}_H]_{t_0 + \delta_1} \qquad (\delta_1 > 0)$$

Note that $\underline{ep}_{Bob} \cap \underline{ep}_q$ is not empty †. If we introduce ep* to denote all objects potentially available through a given ep (e.g. globals) then $Q \in \underline{ep}^*_{Bob}$ or $Q \in \underline{ep}_{Bob}$. It is possible that Q itself is the only connection between $\underline{ep}^*_{Bob}$ and $\underline{ep}^*_Q$, in which case BOB calls Q to achieve either changes in $\underline{ep}^*_Q$ or $\underline{ep}_Q$ (most useful in systems allowing general goto's). The most common case, however, is that $\underline{ep}_{Bob} \cap \underline{ep}_Q$ contains in addition to Q some data object whose value is changed as a result of Q's execution.

Assuming therefore that $\underline{ep}_{Bob} \cap \underline{ep}_Q \neq \emptyset$ and that the execution of Q has yielded some non-trivial result, then upon return to BOB we have

---

† The set operations $\cap$, $\subset$, $\cup$ when operating on ep's are interpreted as operating on the (possible) common variables of the two ep's.

$$\text{BOB: } [\underline{ep}'_{Bob},\ \underline{ip}_H]_{t_0+\delta_2} \qquad (\delta_2 > \delta_1)$$

where the "'" denotes that some of the variables in $\underline{ep}_{Bob}$ have received new values. At this point, we will drop the time designation, since we are dealing with sequential programs and since we assume that time is increasing monotonically. We now consider the situation where H is in fact not a "real" hardware machine, but is implemented by an interpreter called J (e.g. H is microprogrammed on a J-machine). Thus we have

$$\text{H: } [\underline{ep}_H,\ \underline{ip}_J]$$

The state of BOB's $\underline{ip}$ is completely dependent on the state of H as a whole, and thus we can substitute this last equation in the preceding one, yielding

$$\text{BOB: } [\underline{ep}'_{Bob}, [\underline{ep}_H,\ \underline{ip}_J]]$$

Clearly, if J itself is being interpreted by K, then $\underline{ip}_J$ can be replaced in the same fashion, ad infinitum until the true hardware execution level is reached, i.e.

$$\text{BOB: } [\underline{ep}'_{Bob},\ [\underline{ep}_H,\ [\underline{ep}_J,\ [\ldots\ [\underline{ep}_Z,\ \underline{ip}_{host}]\ldots]]]]$$

## 2.2 Contemplation

If we now contemplate this last expression, there are several observations we can make.

1. $\underline{ip}_{host}$ cannot by definition be further decomposed, which corresponds nicely with the traditional view of the $\underline{ip}$ as a simple integer program counter and nothing more. It is also appropriate to the common situation where the true host hardware is simple in the extreme. A more practical viewpoint however is that the location of the $\underline{ip}_{host}$ level is a design decision reached by concluding that any additional $\underline{ip}$ fine structure is irrelevant to the task at hand e.g. none of the interpreters below this level are explicitly "invocable".

2. It is useful to postulate an abstracted pointer to the program's code, cp, which is not PC but merely a reference to where the code is to be found. The distinction between ep and cp is necessary if the dual requirements of reentrancy and protection are to be fulfilled. Thus it is most often found in newer systems and most often blurred in older ones, where $cp \subset ep$ or at best $cp \subset ep^*$. However, the opposite extreme, $cp \cap ep^* = \emptyset$ must somehow also be avoided, since otherwise there is no way for the code to reference the ep it is to manipulate. This conflict is resolved by the code's assuming a particular mapping (e.g. linear or tree-structured) which can be expressed in terms of integers. Since integers are not tainted by differing name environments, it is only required that the ep supplied to the code conform to the same mapping which is assumed by the code. The breakdown of this mapping is commonly called a 'bug' i.e. the code is manipulating the ep, but the ep's structure does not corespond to that assumed by the code. This state of affairs manifests itself as either "wrong answers" or a machine semantic error, or both.

3. Expanding on the above, it is the interpreter which ties the code to the program ep. It fetches instructions and (integer) addresses (or address displacements) from cp and applies the resulting effect to ep. Thus the activating call on an interpreter must include access to both $ep_{program}$ and $cp_{program}$. Therefore, cp belongs in the realm of the interpreter, and not in that of the program, and the temptation to write P: [ep, ip, cp] as the characterization of an incarnate program should be resisted.

4. The state of (e.g.) BOB is contained in two places, $EP_{Bob}$ and $EP_H$; $EP_{Bob}$ holds the variables generated by the execution of BOB's code, and $EP_H$ holds (generally speaking) the variables determining the next instruction to be executed. In analogy with the "where is $PC_{Fred}$" problem, a related phenomenon is the pushed-down values of PC (i.e. the "return" stack), which are

most often kept in the program's EP. This information is totally incomprehensible within the context of this EP, and one carefully avoids manipulating it. That such an allocation is so common today is probably due to the fact that (hardware) machines have historically been restricted by economics to have minimal ep's, and therefore the program's ep was a very convenient place to stash such information. However, in the context of our discussion there is no longer any reason to continue to mix such apples and oranges together, i.e. the return stack belongs in the interpreter's EP.

## 3. THE CROSS-INTERPRETER CALL

In any computer system, a useful requirement to place on any given program is that its caller need not know anything about the environment in which the called program runs. In a system with multiple interpreters, this means that the caller need not know what machine the program runs on, nor anything about the depth of the possible nest. To achieve this, we have chosen to expand Landin's closure model [9]. Figure 3 illustrates the earlier example of program FRED, and should be interpreted as the static [i.e. pre-invocation] data structure describing FRED's necessary environs. The job of a Cross-Interpreter Call of FRED is to erect the data structure shown in Figure 4. The important features of Figure 4 are (1) the closure is unchanged i.e. contains no information about this particular activation of FRED and thus can be reused in other calls, and (2) the cp's and PC's of each level are referenced solely through the local environment of the next lower level.

An algorithm for accomplishing the cross-interpreter call appears in Figure 5 and it should be noted that any member of a nest may validly invoke it. It should be emphasized that this code body is in principle not a procedure, since "calling" it as such would result in an endless loop of calling itself to "call" itself. Rather, this code must operate outside of the semantic space it manipulates, and is "god-given" as far as all other entities in the system are concerned. Other examples of god-given operations are the acquisition and return of resources, and, in the last analysis, the reading and writing of physical storage. Thus that set of operations made available by the host machine is identical to the set of god given operations. Some of these operations may be 'wired into' the host machine, while others are composed of sequences of such hardware primitives, but their appearance to the upper levels is the same.

### 3.1 Linkage Mechanisms

Note that the Cross-Interpreter Call mechanism described in Figure 4 says nothing about the way Fred's caller passes its parameters to Fred. Since the process of transferring parameters can become arbitrarily complicated, it is important that this complexity not be a part of a basic system operation. Furthermore, the option of different parameter passing conventions would thereby be excluded.

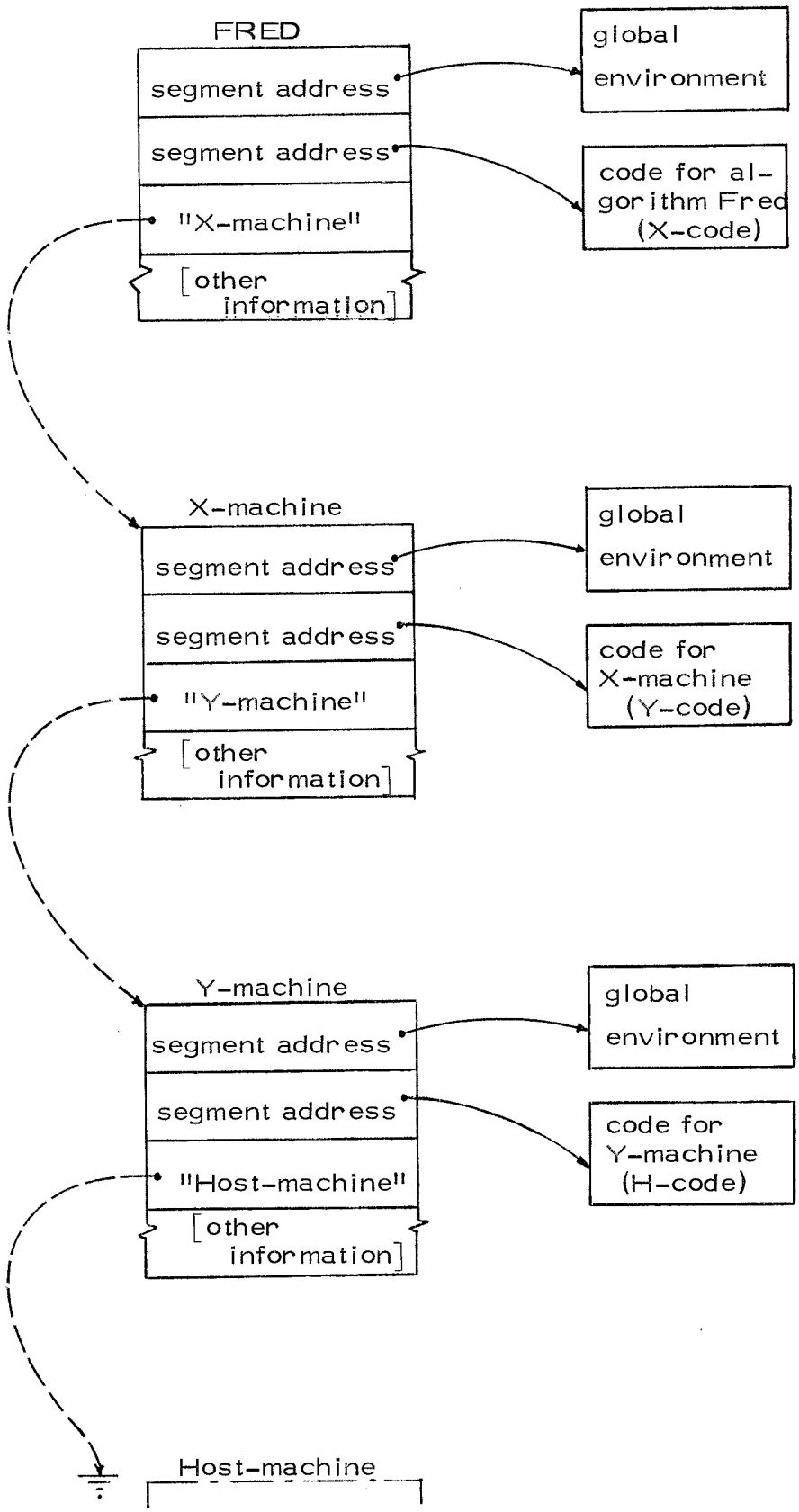



Fig. 3. FRED's Static (ep, ip) Structure - Hard links between the levels will be established via the Cross-Interpreter Call mechanism, and are here considered to be strings to be mediated via the File system.

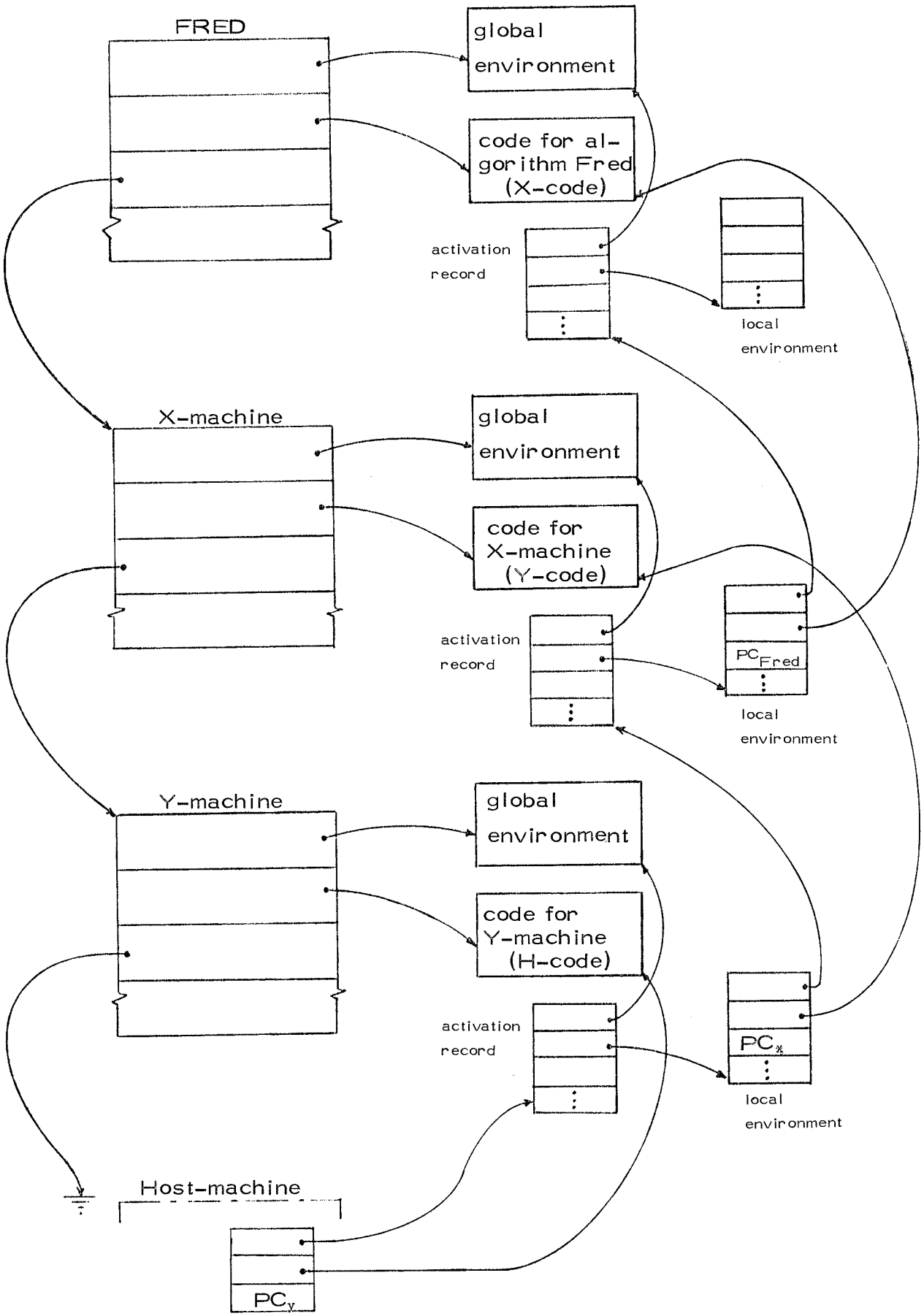



Fig. 4. FRED's Dynamic (ep, ip) Structure

- All pointers are transformed by the Cross-Interpreter Call into segment addresses.

```
Cross Interpreter Call (closure) =
{
    comment erect the data structure of Figure 4;
    local actrec, locenv, codeptr;
    comment set up environs for the procedure described by "closure";
        setuplocalenvironment; comment locenv now points to the new local environment;
        setupactivationrecord; comment actrec now points to the new activation record;

    comment set up the environs for all the interpreters required by the called procedure;
    while ip.closure ≠ hostmachine do
    begin
        codeptr := cp.closure;
        closure := ip.closure;
        setuplocalenvironment;
        locenv.cpslot := codeptr;
        locenv.pcslot := 0; comment there must be a system-wide convention for this value;
        setupactivationrecord;
    end

    comment finally, save the caller's state and transfer to the callee;
        savecurrentstate (restarthostPC, hostregister1, hostregister2);
        hostregister1 := actrec;
        hostregister2 := locenv;
        hostPC := 0;     comment op.cit;

    comment at this point, the deepest interpreter in the nest starts running
        on the host, and consonant with the bottom-up control regime, the
        upper interpreter levels are started, until finally the called pro-
        cedure begins 'execution'.
}
```

Fig. 5. An Algorithm for Cross-Interpreter Call - The procedures setuplocalenvironment, setupactivationrecord, and savecurrentstate are considered local to the algorithm. This algorithm handles only a normal procedure call without parameters. See the text for a discussion of other possibilities.

The most straightforward conventions apply when parameters need no evaluation in the callee's environment, and hence the caller can arrange to acquire what will later be Fred's activation record and compute all parameters. A reference to this activation record is then an additional parameter to the Cross-Interpreter Call. If however Fred must cooperate in the evaluation of its own parameters, then the caller must still acquire a resource, in this case a means for them to synchronize with each other. Fred is then set up and able to participate in the dialogue. Whether this dialogue is viewed as inter-process communication or a co-routine relationship is dependent partly on philosophy, and partly on the linkage mechanism actually used. Lampson et al [10] and Kahn and MacQueen [25] describe general linkage mechanisms which are applicable to these problems.

As an example of the power of generalized linkage mechanisms, it is possible to subsume the ordinary procedure call by viewing it as a primitive form of synchronization between the caller and the callee, wherein the caller 'waits' on a common event variable until the callee signals via same that it is finished. This arrangement allows the caller, if it so wishes, to spawn a number of procedures as (more or less) independent processes, and itself to continue to execute. A second example is the above mentioned co-routine based parameter passing protocol.

In closing, we take up the subject of returning from the callee to the caller. At each level of the nest, superfluous resources must be released (e.g. memory, via appropriate god-given operations) before executing a Cross-Interpreter Return, and this sequence of actions is repeated at each level of the nest. When 'control' arrives at the bottom of the nest, the state of the caller's host machine is retrieved from the place where the Cross-Interpreter Call saved it, and when this information is restored to the host machine, the caller will resume executing at the point immediately following the call (in the case of a simple procedure call). This simple restart is a consequence of the bottom-up control regime mentioned in the introduction.

The problem of the callee's returning a value (e.g. Fred is a function procedure), like that of passing parameters, is the responsibility of the caller and the callee and not of the Cross-Interpreter Return.

The reader should not form the impression, however, that all procedure calls and returns in a multi-interpreter system are accomplished via the god-given cross-interpreter call/return mechanisms. Clearly, for those procedures which run on identical nests, this would entail excessive and unnecessary overhead. If such procedures are compiled as a unit, for example, then the compiler has sufficient information to generate code for the language's 'internal' procedure linkage. It is only in the case when a procedure is 'external' that it is necessary to use the system-defined linkage. Whether this linkage is then to an identical or completely different ip environment is, for better or worse, unimportant.

### 3.2 Multiprocessing and Multiprogramming

The focus of this section is the ip field of a procedure's closure. This field can in theory have three types of values:

1. a (file system) name of an interpreter program
2. a (possibly generic) name of some physical (host) processor
3. a pointer to the activation record of some interpreter

We have heretofore only discussed the first of these, with the result that each time a given machine was required, a new incarnation of it was created via its closure. This can be accepted without discussion except in the case where the given machine is the host-machine, in which case we are in general specifying a multiprogramming of the host. If we did not wish this to happen, then one possibility is to interpret the ip name in the closure not as a software machine, but as a hardware machine. In this case, it would be the job of the cross-interpreter call to determine if the name is that of one of the (currently unallocated) host processors available to the system, and if so to activate the nest on that processor. There is no reason, based on our discussion, for these additional host processors to be either physically or functionally identical to the single 'host' currently executing the cross-interpreter call.

The result of this logic is the possibility for systems having a large number of physical processors, some being functional copies of other, others being fundamentally different. Contemporary hardware developments make such a system economically possible, while the cross-interpreter call mechanism

suggests an automatic means of allocating the individual processors. Such a system, when based on a generalized transfer mechanism, displays true parallelism at the procedure level, a type of parallelism which is neither as fine-grained as that of pipelined processors nor as coarse as that of a 'job'.

Returning to our main thread of discussion, the third possibility for the ip field of a closure is a pointer to the activation record of some interpreter. The very existence of an activation record means that the interpreter is already incarnate, and thus an attempt to graft a new nest upon it is in fact a request for multiprogramming. If such a possibility is admitted, then the cross-interpreter call must interrogate the incarnation concerning its willingness to permit multiprogramming.

It should be apparent that this kind of handshaking could become rather involved, especially since at least one of the parties to the conversation is the (presumably primitive) host machine. The only case where this possibility might find application is if the callee is an encapsulation of some existing system (e.g. IBM 360/370, DEC PDP-10 etc.), but even here it is not immediately apparent that a software-based multi-incarnation system would not be easier to deal with. The only exception to this logic is the case where the already incarnate machine is a host-level processor, and what is perhaps the ultimate in generality is achieved by allowing this possibility.

### 3.3 Input/Output

The ability to have multiple but functionally dissimilar physical processors has special relevance to input/output. On the one hand, I/O by its very nature invokes the presence of a second parallel process; the true parallelism of this process to the calling process has been made increasingly visible in contemporary systems by the introduction of ever more sophisticated and independent data channels, sometimes even as distinct physical processors. On the other hand, there has been a software need to reflect the underlying synchronization mechanisms in this mutual cooperation, which need is difficult to accomplish in practice. The use of a general cross-interpreter procedure call of the type described in § 3.1 supplies the necessary synchronization mechanisms while still allowing the I/O to be carried out by a separate processor.

## 3.4 Virtual Machines

It is natural to assume that the concept of a virtual machine [ 18] is closely connected to that of an interpreter. In this section we analyze virtual machine architectures using the machinery developed in the preceding sections.

A virtual machine can be considered a copy of the same machine which all other software on a system runs on i.e. there is only one interpreter per se in the system, but for each virtual machine which is erected, there exists an additional incarnation of the interpreter. The simple case of the activation of a single virtual machine is shown in Figure 6. The virtual machine is supervised by the Virtual Machine Monitor (VMM) whose job it is to maintain the virtual machine's illusion that it is a real machine. This means that VMM is most frequently activated by the execution on the virtual machine of instructions which are normally considered 'privileged' e.g. I/O, page/segment table updates, etc. When such an instruction is attempted, the virtual machine must activate some code in the VMM (and in the VMM EP)which provides the necessary simulation of same.

Thus the sequence of events in the life of a virtual machine is

1. A cross-interpreter call to the 'user program' from the VMM, which yields a new incarnation of the system machine.

2. Start of execution of the user program.

3. VMM is activated by the virtual machine when certain privileged instructions are attempted.

4. Control returns to the virtual machine when VMM completes each instruction simulation.

5. Continued execution of the user program. (Steps 3, 4, 5 are repeated an arbitrary number of times.)

6. The user program 'returns' to its caller (the VMM) and the virtual machine is destroyed.

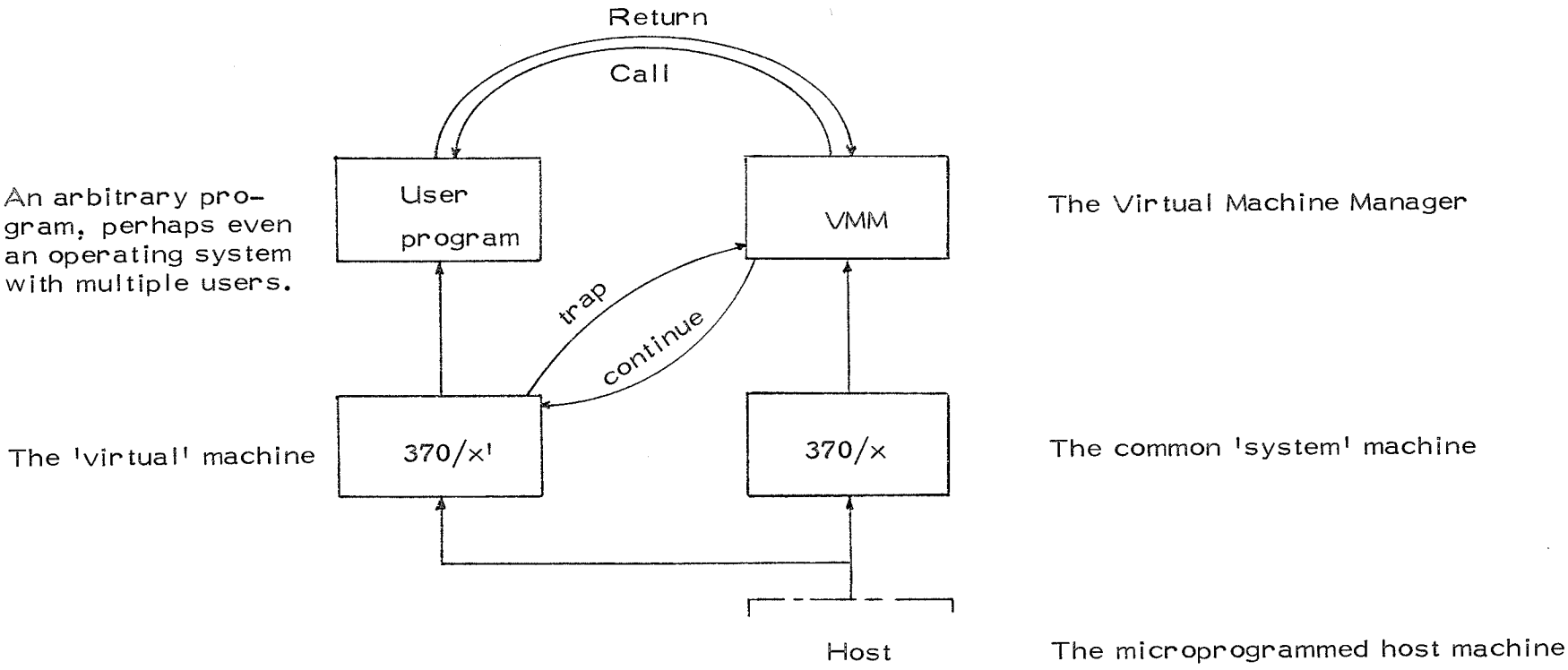



Fig.6 . The Structure of a Virtual Machine Activation - Additional (non-recursive) virtual machines have the same structure as the left-hand component, and share the same VMM.

Items 3 and 4 above are a distinct (external) transfer of control from one interpreter regime to another. The question, however, of what kind of transfer of control is used is a design choice. One possibility is to view the interaction as a co-routine relationship which fits nicely with the fact that control is passed to an already existing environment. On the other hand, if this environment is such that all the relevant information is in some sense global, then a procedure call from the virtual machine (program) to a routine in $(\underline{ep}, \underline{ip})_{VMM}$ is sufficient.

An important development for virtual machines is the possibility of their recursive activation [18, 19], i.e. the possibility of creating 'nests' of virtual machines, each of which supports the virtual machine 'above' it. The additional criterion is placed on such nesting that all virtual machines in the 'nest' execute equally fast. Figure 7 shows that such a 'nest' of virtual machines is not a nest in our sense of the word, and therefore the exponential drop in speed associated with interpretation nests applies only in exceptional cases.

In closing, we note that our model concerns itself with environments and the flow of control between them. Therefore, data structures internal to the processes (such as the $f$ and $\varphi$ maps treated by Popek and Goldberg [17]) lie outside the present discussion.

### 3.5 Dynamic Machines

We define a dynamic machine to be one whose repertoire of instructions as seen by the interpreted program changes in time. Such a machine is therefore useful when designing an interpreter for a language which allows the dynamic definition of new data types and (in particular) new operations on these types.

When a new operator is introduced (usually on a block boundary) in a language, its effect is defined by a procedure written in that language. The question now is to decide to which target machine the procedure should be translated. The traditional approach has been to execute the procedure on the same target machine as the rest of the program, but clearly some execution speed is lost by introducing this additional level of interpretation. The advantage of the scheme is its simplicity: nothing new is expected of either compiler or machine.

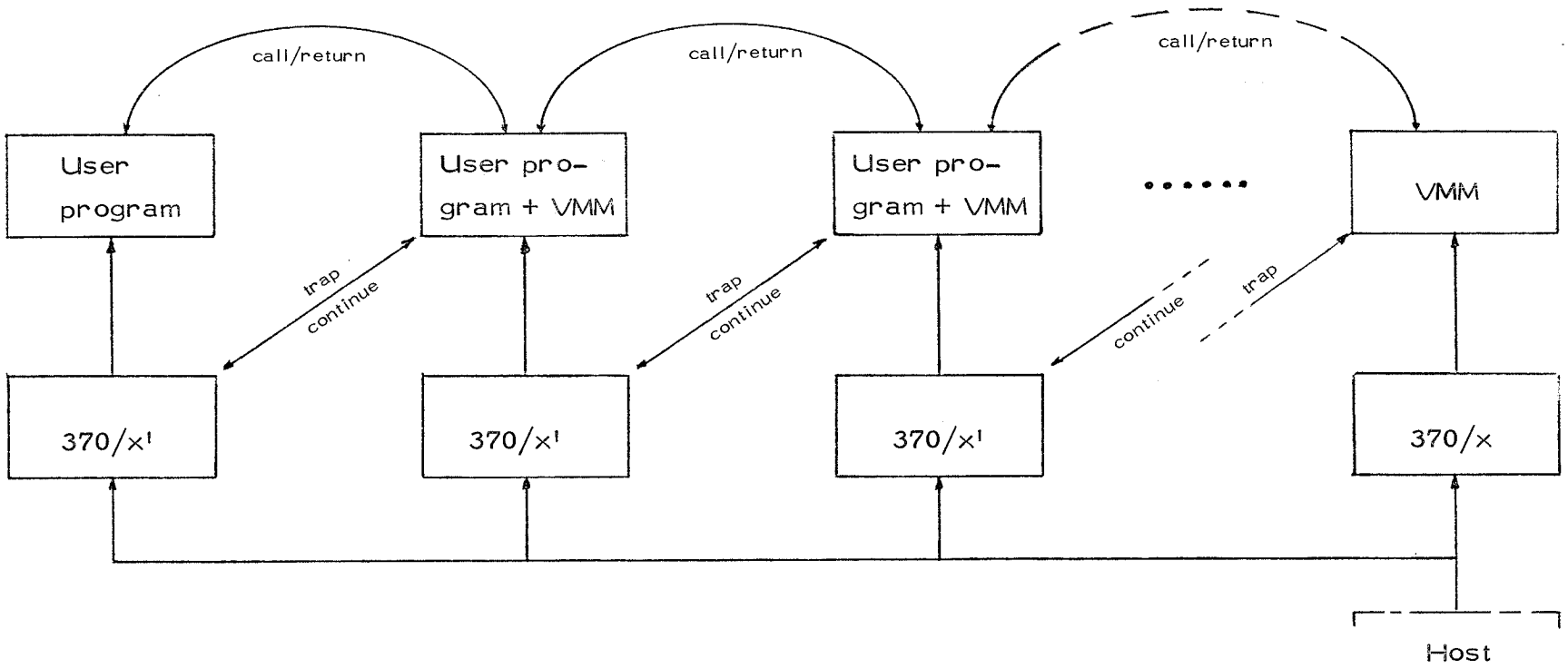



Fig. 7. The Structure of a recursive 'Nest' of Virtual Machines

- Note that this is not an interpretive nest in our sense of the word. The case where a fault at the 'outermost' level causes faults at successive lower levels all the way back to the 'bottommost' VMM is viewed as the activation of a series of procedures in differing (ep, ip) environments. The telling point is that the successive 370/x' machines could in fact all be very different from each other without changing the conceptual basis for implementing recursive virtual machines.

The 'bottom' of the 'nest'.

A second approach to the problem is to execute the operation-procedure on some other, presumably faster, machine. This other machine can then be e.g. the language's target machine's machine, some other machine which offers appropriate advantages, or the host machine. If the operator procedure is to run directly on the host machine, then it is important to realize that it in general should not exist as a (host-) instruction, but rather as a Callable procedure. The reason for this is that if it is an instruction, then it is available for execution only at the bottom of whatever nest it appears in. This in turn means that the access path to it is through all the layers of the nest i.e. the top level executes instruction NEW, which is interpreted at the next level by executing NEW', which the next level interprets by executing NEW'' etc. until finally the bottommost level is reached and the new operation can actually be carried out on the host. If on the other hand NEW exists as a procedure, a Call on it filters down through the nest just as NEW did, but ends in a call of the function. Thus the trade-off is between the overhead of the Call mechanism and the overhead of modifying and executing multiple layers of interpretation. It is important to realize that these are the only possibilities, and that there is no way to cheat by somehow executing directly on the host from some arbitrary level (vague claims in the trade press notwithstanding).

If the operator procedure is executed on some machine which is not a member of the nest, then a procedure invocation is the only choice, and the environmental modifications which must take place are heavily dependent on the degree to which the binding of the new god-given operation to the machine which will execute it is delayed. In the simplest case, separately compiled instructions are disallowed, thus enabling the language compiler to assign a unique cardinal value to each. A new version of the new instructions' common target machine is then compiled, and a reference to its closure is made available in the original program's EP. Finally, the language's target machine must have an instruction with two parameters: the aforementioned closure and the extended operation's cardinal value. The latter is used by the language target machine as a parameter (together with the current program ep) in a Cross-Interpreter Call on the former. Less simple cases (e.g. where a more delayed binding is allowed or multiple 'external' machines are required) are correspondingly more complicated and perhaps deserve further research.

This discussion of dynamic machines is based on the Call mechanism introduced earlier, which in turn is based on the information contained in the extended closure. We have chosen this method because of its simplicity. The apparatus introduced in [11] contains dynamic machines more clearly, but is correspondingly less efficient. Furthermore, this is the only instance we have found in our investigations where the extended closure is in some sense insufficient.

# 4. OVERALL SYSTEM CONSIDERATIONS

In this section we consider the more global implications of a system built on the principles described in the two preceding sections.

## 4.1 Storage Management

We mentioned in the introduction that an activation record should be allocated as a segment. An immediate implication of this is that a general multi-interpreter system will require a large number of segments, but that in most cases the maximum segment size need not be great. Of much greater interest than their number and size, however, is how these segments are organized. The goal of any such organization is to minimize, for reasons of both protection and efficiency, the number of segments available at any given time to a computation. A means of doing this is to exploit the well-known locality of computations i.e. the range of memory access is usually a small subset of the total possible at any given instant. Thus the locality of execution is associated with the activation records of the interpreters in the nest plus the activation records of the program at the top of the nest. The contents of these activation records are closely connected to the static structure of each of the programs, and thus what we desire of our segment organization scheme is the recognition of this static aspect.

Fortunately, a segment management scheme which models the static locality of data access is available: the domain concept [20, 21, 22, 23]. Domains were originally introduced as a protection mechanism, but it has since been realized that they can also be viewed as the static component of a program's working set. A computation moves from one access domain to another either via a jump or a domain "call" (in which case the domains can be recursively nested). A domain corresponds to a segment, which in general may contain other domains (segment descriptors). There is such a natural correspondence between this data structure and that of a program's activation records that one can profitably view procedure invocation as being replaced by domain invocation. The locality of the computation becomes explicit therefore in every domain change, with resultant gains by the resource management functions of the system. The conclusion we make on the basis of this discussion is that the necessary glue to hold a multi-interpreter system together is provided by the domain concept, and that it is therefore wise to view the cross-interpreter call as an augmented domain-change operation.

With these considerations in mind, as well as those of Section 3.2, a natural analogy arises to domain-based multi-processor systems such as Hydra [22], in which connection it should be noted that the structural theory is identical to the present paper's (LNS ~ ep, process ~ EP).

## 4.2 Structure

We have, in the preceding sections,devoted our attention to nests of interpreters, the primary reason being to ensure that any conclusions regarding degenerate nests (depth = 1) were properly drawn. Because of the exponential rise in execution time with increasing nest depth, it is important to be aware of the trade-offs involved in choosing between a procedure structure and an interpreter structure for a given application.

The traditional use of interpretative execution is to implement programming languages whose level of abstraction is far above that of the host machine. Thus an abstract machine suitable for the language is designed and implemented via an interpreter, thereby reducing the complexity of the compiler to a manageable level. The result of this approach, besides implementation convenience, is usually a trade-off between speed and memory (see Table 1). It is also illuminating to examine those cases where there is improvement both in speed and memory.

The procedure call and Ifetch mechanisms can be viewed as duals of each other, in that the environment where further instructions are executed is different for each "instruction invocation". In the case of procedure call, the PC is saved and changed, and the current $EP_{PROG}$ is augmented to include the ep of the newly entered procedure. In the case of an ifetch, the PC is updated and execution 'proceeds' in the already existing ep of the interpreter. Given that augmentation of the ep is a more complicated operation than merely fetching the next instruction, we can see that the Ifetch mechanism is fundamentally simpler than that of a procedure call. This advantage and the fact that the ep's of the nest-components are set up once and for all at nest-activation time (and are therefore essentially static) are countered by the Ifetch at each level being interpreted by the next lower level and therefore plagued by exponential slowness.

| Language | Implementation Approach | Host Machine | Relative Object Program Size | Relative Object Program Speed | Reference |
|---|---|---|---|---|---|
| BCPL | closed subroutines | Nova | 1.0 | 1.0 | Veie [15] |
| | threaded code | Nova | 1.2 | 1.5 | |
| | 0-code interpreter | Nova | 0.57 | ~ 3 | |
| STAB-1 | straightline code | PDP-11 | 1.0 | 1.0 | Colin et al. [16] |
| | STAB-12 interpreter | PDP-11 | 0.47 | 3.3 | |
| Fortran | straightline code | PDP-11 | 1.0 | 1.0 | Bell [3] |
| | threaded code | PDP-11 | 0.8 - 0.9 | 0.97 | |
| RPG, Fortran and Cobol | Microcoded Interpretation | B1700 | ~ 0.5 | 0.08-0.5 (!) | Wilner [1, 2] |

Table 1 Results of Various Languages Implementation Strategies

Interpretation usually yields a more compact but slower implementation; the B1700 results are due to careful analysis and design, as mentioned in the text.

Thus the key to producing an interpreter which can compete in speed with a direct host machine implementation is to minimize the Ifetches at each level. This means that (1) the primitive operations at each level must be carefully chosen to get the maximum yield for each Ifetch, and (2) no operation which can be performed at a lower level should be performed at a higher level. It is particularly the latter which is decisive, and the fact that the B1700 [24] host can perform varying length arithmetic means that the interpreter need only pass such operations down to the next level. Thus the 'anomolous' results obtained by Burroughs are explainable by application of the above two optimization principles. Nevertheless, it is doubtful that nests of depth > 2 ever can yield greater speed than direct host execution, and the problem of trading speed off against memory thus again rears its head.

A more fruitful application of nests is where the primary goal of execution is not computation but rather control. A good example of this is job control languages. The abstract machines for such languages contain instructions concerning sequencing through job steps and manipulating the job's EP at a very high level. The most important goal of such a language is to supply the kind of sequencing required by real-life jobs, and the raw speed of accomplishing this is less of a factor at the macroscopic job-level. The fact that such facilities as arithmetic and conditional statements, loops, local storage, general parameterization, and multi-job synchronization are usually unavailable in job control languages hangs together with the viewing of the problem through procedure-glasses. An interpreter, and not procedures, is the natural vehicle for the implementation of more flexible and usable job control languages (e.g. ICL's System Control Language [14]). This approach, at a nesting level of one or two, should offer the same (slow) speed, but permit better facilities and a cleaner structure.

Our final example of the use of nests concerns the "idle loop" in a multiprogrammed operating system. One would ideally choose to execute a synchronous Wait until some process appears in the Ready queue, but this leads back to the original problem of the Ready queue's already being empty. If the system is highly structured, finding a suitable ep in which to idle can thus be a nasty problem. The traditional solution is to define an Idle process which appears only when this situation arises; such a process provides the problematical ep but involves extra bookkeeping to keep it out of the way when the Ready queue is not empty. An alternative solution is to provide the Wait function as

an (interpreted) instruction instead of as a procedure. This device moves the idle-loop from the operating system's EP to the interpreter's EP where (given the latter's simplicity with respect to the former's) it is more convenient to idle. More significantly, the body of the Wait function, by virtue of its being a single 'instruction', is automatically a critical region with respect to the level at which it was invoked (another consequence of the bottom-up control regime). This fact, coupled with the static nature of an interpreter's EP, makes the earlier analogy to Simula classes and monitors even more relevant, and suggests the implementation of monitors, or their equivalent, as interpreted functions. In summary, it would seem that the interpretation of operating system primitives at a lower level offers new means for structuring the solutions to traditional operating system problems.

## 5. CONCLUSIONS

We have presented a model for multi-interpreter systems which describes the control structures necessary to allow programs which run on one machine to invoke programs which run on other machines. These machines may be host-level processors or defined by programs which run on still other machines. The expansion of the model to describe systems with multiple (not necessarily identical) physical processors is straightforward, and the extent of parallelism obtainable depends on the generality of the linkage mechanism chosen. In either case, a domain-oriented storage management systems is a natural symbiosis.

Interpreter-based systems display structural attributes which are different from procedure-based systems and offer therefore implementation advantages when code-compactness is important, when a more suitable level of abstraction is required (e.g. for code generation), when computation per se is not the primary goal of execution, and when exception conditions arise which are philosophically incompatible with the EP in which they occur (the 'idle-loop' problem).

This paper has not addressed the problems of common data formats which arise immediately in multi-interpreter systems, nor (the implementation of) less general but philosophically compatible systems. The interested reader is referred to Derrett [12] and Lynning [13] for further discussion of these matters.

## 6. ACKNOWLEDGEMENTS

I would like to acknowledge Nigel Derrett's sine qua non midwifery of these ideas; and Nick Shelness' very helpful discussions of protection domains.

REFERENCES

[1] Wilner, W.T., The Design of the B1700. Fall Joint Computer Conference, AFIPS, 1972. pp. 489-497.

[2] Wilner, W.T., B1700 Memory Utilization. ibid pp. 579-586.

[3] Bell, J.R., Threaded Code. CACM 16, 6, June 1973.

[4] Hoare, C.A.R., Monitors - An Operating System Structuring Concept. CACM 17, 10, October 1974.

[5] Organick, E.I., Computer Systems Organization - The B5700/B6700 Series. Academic Press, New York, 1973.

[6] Johnston, J.B., The Contour Model of Block Structured Processes. Proc. of a Symposium on Data Structures in Programming Languages, Ed. Tou and Wegner, ACM/SIGPLAN, February 1971.

[7] Wegner, P., Programming Languages, Information Structures, and Machine Organization. McGraw-Hill, New York 1968. pp. 8-23.

[8] Rosin, R.F., Contemporary Concepts of Microprogramming and Emulation. Computing Surveys 1, 4 p. 208-210, 1969.

[9] Landin, P.J., The Mechanical Evaluation of Expressions. Computer Journal 6, pp. 308-320. 1964.

[10] Lampson, B.W., Mitchell, J.G., Satherwaite, E.H., On the Transfer of Control Between Contexts. Lecture Notes in Computer Science, Ed. Hartmanis and Goos. Vol. 19. (Programming Symposium in Paris, April 9-11, 1974.)

[11] Manthey, M.J., Nested Interpreters and System Structure. DAIMI PB-51, September 1975. Computer Science Dept., Aarhus University, Denmark.

[12] Derrett, N.P. and Manthey, M.J., Multi-Interpreter Systems. DAIMI PB-55, January 1976. Computer Science Dept., Aarhus University, Denmark.

[13] Lynning, E., A Multi-Emulation System. DAIMI PB-62, July 1976. Computer Science Dept., Aarhus University, Denmark.

[14] Barron, D.W., Job Control on the ICL 1900 Series. Computer Bulletin. March 1976.

[15] Veie, O.C., The Implementation of High Level Languages on Mini-computers - A Case Study of BCPL on the NOVA. Computer Science Dept., Aarhus University, Denmark 1977.

[16] Colin, A.J.T., Shorey, K., and Teasdale, W. The translation and Interpretation of STAB-12. Software-Practice and Experience. Vol. 5, p. 123-138, 1975.

[17] Popek, G.J. and Goldberg, R.P., Formal Requirements for Virtualizable Third Generation Architectures. CACM 17, 7, July 1974.

[18] Goldberg, R.P., Architecture of Virtual Machines. National Computer Conference AFIPS 1973. pp. 308-318.

[19] Buzen, J.P. and Gagliardi, U.O., The Evolution of Virtual Machine Architecture. ibid. pp. 291-299.

[20] Fabry, R.S., Capability-Based Addressing. CACM 17, 7, July 1974.

[21] Dennis, J.B. and van Horn, E.C., Programming Semantics for Multi-programmed Computer Systems. CACM 9, 3, March 1966.

[22] Wulf, W. et al. HYDRA - The Kernel of a Multiprocessing Operating System. CACM 17, 6, June 1974.

[23] Needham, R.M. and Wilkes, M.V., Domains of Protection and the Management of Processes. Computer Journal 17, 2.

[24] Salisbury, A.B., Microprogrammable Computer Architectures. Elsevier 1976.

[25] Kahn, G. and MacQueen, D., Coroutines and Networks of Parallel Processes. IRIA Report 202, Nov. 1976.

[26] Rosin, R.F., Frieder, G., and Eckhouse, R. A Research Environment for Microprogramming and Emulation. CACM 15, 8. August 1972.