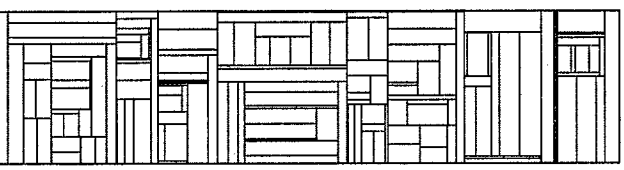
MULTI-INTERPRETER SYSTEMS

by
Nigel Derrett
and
Michael J. Manthey

DAIMI PB-55
January 1976

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



ABSTRACT.

This paper is concerned with systems to support high-level-language-oriented interpreted machines. Certain requirements for such a system are presented and some of the problems which arise are discussed.

Keywords and Phrases.

Interpretation, emulators, microprogramming, high-level language implementation, procedure call, machine architecture.

CR Categories.

1.3, 4.13, 4.29, 4.30, 4.35, 6.20.

CONTENTS

1. INTRODUCTION.

The authors' aim in this paper is to present some thoughts as an introduction to what we believe to be an interesting and largely unexplored topic in computing. If we can stimulate others to venture forth into it we shall have achieved our goal. The interested reader is also referred to [1].

If one wishes to implement a high level language, then doing so on a machine designed with that language in mind can have considerable advantages over using a more general purpose machine [e.g. 2]. Such a high-level-language-based machine can be implemented directly in hardware [Burroughs B6700, 3] or by a software interpreter [e.g. Burroughs B1700, 4].

In this paper we shall consider what sort of system should be provided to support programs written in various high-level-languages runnning on their respective interpreters. Existing systems (such as the B1700) usually have three well-defined levels:

Programs, written in high level languages;
Interpreters, written in machine code;
Hardware.

(Whether or not the hardware is called "microprogrammable" is not very important here. It should be suitable for writing interpreters, which tend to be small and repetitive.) We believe this three-level hierarchy to be somewhat restrictive and shall consider a more general arrangement where interpreters need not be written in (hardware) machine code.

It may be helpful here to clarify a few terms. We shall use the words "interpreter" and "emulator" more or less interchangeably, although we shall tend to use "emulator" when referring to an interpreter for a specific (often hard-wired) machine (e.g. a PDP10, [5]). The rest of this paper will be written in the assumption that we are trying to design a system which can support programs running on interpreters in a general way. This system would be based on some hardware, which we shall call the "host machine".

2. REQUIREMENTS FOR A "GENERAL" SYSTEM

The introduction of several machine codes, with corresponding emulators, into a system adds an extra level of complexity to the whole; and, further, assumptions which can be made about a single-machine-code system are not always true for a multi-code one. Thus the authors found it helpful to have some sample requirements in mind when thinking about the system.

In this section we shall state some problems which a "general-purpose multi-interpreter system" should be able to cope with. We shall give some motivation for them in order to persuade the reader that they are reasonable. In the next section we shall consider some of the difficulties which arise when we try to fulfil these requirements.

Requirement 1

A program running on one emulator should be able to call a procedure which runs on another.

This allows, for example, an I/O package written in one language to be used by the others. This is important, not only because it saves writing an operating system for each interpreter, but also because it allows the system to be much more secure. (For example it is possible to have only one interpreter which can deal directly with the disc file-system.)

Requirement 2

It should be possible to write a procedure in host machine code (microcode) and call it from a program running on an interpreter.

This allows fast execution of time-critical operations.

Requirements 1 and 2 together state that it should be possible to write and call procedures which run on any machine, interpreted or host.

Requirement 3

It should be possible to write an interpreter in a high level language.

In other words the system should be able to deal with interpreters which are themselves interpreted. This is quite a difficult requirement to fulfill, for once we lift the restriction that an interpreter must be written in host machine code then we must consider the possibility of "interpreter nests" of any depth. Suppose procedure Fred runs on an X machine, which is emulated on a Y machine, which is emulated on a Z machine, which is emulated on the host machine (see Fig. 1). Then when Fred is called, the system must find, or produce, an X machine for it to run on, and hence Y and Z machines; and all these machines must be set running properly.

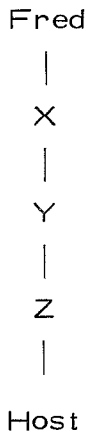


Fig. 1

Requirement 4

It should be possible for an interpreter to call a procedure which runs at any level of interpretation.

Thus, for example, if an interpreter contains a complicated operation (sych as a garbage collector) this could initially be written in a high level language for testing and later written in (for example) host machine code in order to speed it up.

If requirements 1-4 are satisfied then it should be possible to have both programs and interpreters written in a mixture of machine codes. Thus the first version of either can be written in the most convenient language and then transferred as required, one part at a time, to a more efficient implementation.

Requirement 5

The calling sequence for a procedure should be independent of the interpreter nest on which it runs.

Suppose procedure Fred exists in X-code, and the X-code interpreter is written in Y-code and the Y-code interpreter runs on the host machine. Then, as stated above, in order to call Fred we have to set the X and Y interpreters in motion. Suppose now that the X-code interpreter is rewritten to run on some other machine (e.g. the host machine); then it should not be necessary to change the way in which Fred is called. We should be able to call Fred simply by saying

Call Fred

and it should not be necessary to say

Call Fred on X machine on Y machine

Requirement 6

The representation of a procedure should be, as far as possible, independent of the interpreter nest on which it runs.

Clearly if Fred is written in X-code then it is unreasonable to expect it to run on a Y machine. However the data structure for Fred should be independent of how the X-code interpreter is implemented. Thus, if we consider the example given in requirement 5, the rewriting of the X-code interpreter should not necessitate any changes to Fred. Such a requirement expedites the testing of programs on "private machines" before integrating them into public systems.

Requirements 5 and 6 allow us to make use of the generality provided by requirements 1-4.

We contend that all these requirements are reasonable ones for a multi-emulator system. However, reasonable or not, their fulfillment is rather difficult; we shall consider some of the problems in the next section.

3. SOME PROBLEMS AND SOME SUGGESTED SOLUTIONS

3.1 Common Data Items

The central problem in a multi-interpreter system is the specification of the structures for common data items. Internally, an emulator may use whatever strange formats it likes, but when it communicates with the rest of the system they must use a common notation. When considering what data types interpreters have in common the problem is really which ones to leave out, rather than which ones to include. A possible list is:

Positive integers (indices, sizes);
Characters (for I/O);
Strings (for addressing the file system);
Addresses (e.g. of buffers);
Procedures;
Interpreters.

3.2 Integers, Characters and Strings; wordlength

We do not have much to say about the representations of integers, characters and strings. Some (hopefully not too arbitrary) decision has to be made and everyone must stick to it. It may seem that a common format for integers should be easy to find, for example N bits binary 2's complement, but a Cobol-oriented machine is quite likely to use a decimal representation of integers, and another interpreter might be designed for experiments with non-standard representations. Thus we see that even for integers some interpreters will need to do conversion to and from their internal formats whenever they communicate with the rest of the system.

The question of wordlength arises here, and once again no "nice" solution can be found for a system which may be required to support emulators for a 60 bit CDC 6400, a 16 bit PDP11 and a byte addressed Cobol machine. The common wordlength is quite likely to be determined by characteristics of the host machines (e.g. the sizes of its internal registers).

3.3 Addresses

As regards addresses it is desirable that each interpreter in the system should have its own private data area, which forms its "virtual store";

and in the interests of security we would like to protect this area against interference from other emulators. Thus we could imagine each emulator resolving addresses using its own base and limit registers. On the other hand we may sometimes want to give one emulator access to another's store (e.g. when handing buffers to the I/O interpreter). This is one aspect of the general problem of protection, and in this case a segmented store seems a likely solution - common addresses should contain a segment number and an offset within that segment. We note that the wordlength problem crops up again here - is the offset in bits, bytes or what? and how large is the object prointed to? One might consider putting a "word size" into each segment descriptor. We also note that the items within the segment must be in a format which is common at least to the interpreters which have access to it.

3.4 Procedures

We now come to the matter of the representation of procedures and interpreters; it is this problem which has occupied the authors most.

Whether or not we want to treat procedures as general data items (and pass them as parameters, assign them to variables etc.) it is obvious that we want to create and call them, so there must be some common representation for them. If we think of high level language programs being compiled into special machine codes (and ignore for the moment procedure variables) then trans-interpreter procedure calls can only occur in the case of procedures used as externals in the calling program, and it seems reasonable to insist that they be declared as external (i.e. made known to the system) in the "called" program. If this is so then we can divide procedures (and procedure calls) into two groups:

internal (can only be called from programs running on the same interpreter);
external (can be called from programs running on other interpreters).

Internal procedures can be represented and called in the best way for the given language, but external procedures must have a common representation and calling mechanism. This allows efficient internal procedure

calls, and also frees internal procedures from any restrictions which we may choose to place on external ones.

Some languages (e.g. Algol 60) allow procedures to be passed as parameters to other procedures, and others (such as Algol 68) allow them to be assigned to variables. For such languages an identifier in a program could refer either to an internal or to an external procedure. If these languages are to be introduced into the system and their power used then we must require that their interpreters can differentiate between internal and external procedures. This does not seem an undue restriction on a specially designed machine; it amounts to the interpreter being able to determine whether a procedure lies within its own code area or not.

Thus the use of a common format for procedures seems possible, and we shall suggest one later on, but first we shall consider the mechanics of an external procedure call.

3.5 Parameter Conversion

The passing of parameters from a program to a procedure running on another interpreter causes considerable problems. Suppose program Jim, running on interpreter A, calls a procedure Fred, which runs on interpreter B, and passes it an integer parameter. Then the integer must be converted from A's format to common format and then to B's format. The best place to do this conversion seems to be within the two interpreters. Thus an interpreter must

1. convert all parameters to common format when an external procedure is called (interpreter A's job);

2. convert parameters to internal format on entry to a routine called externally (interpreter B's job).

This means that interpreters must know that they are dealing with external procedure calls and must know the types of the parameters being passed.

It is reasonable to assume that A knows it is handling an external procedure call (we apologise for the anthropomorphic language, but it seems the most appropriate) but interpreter B may not be able to determine whether Fred is being called by a foreigner (in which case the parameter is in common notation) or by a B-code program (in which case the parameter is in B's notation). One way around this problem is to insist that procedures known to the system (declared to be external) always receive their parameters in common format.

There is another, more difficult, problem with the conversion however: how do the A and B interpreters know that the parameter being passed is of type integer? (which they must do in order to perform the right kind of conversion). If A has tag bits for type checking, and if either is a machine for a language with compile-time determinable parameter types (such as PASCAL, but not ALGOL 60) then a solution can be found, but in the case of an untagged machine for a type-free language (such as an OCode emulator for BCPL (6, 7), or an emulator for almost any existing hardware machine) it is difficult to see what can be done unless the programmer somehow "tells" the system what the type is. In any case the addition of tag bits to the common formats so that the types of common data items can be determined at run-time is probably a good idea.

The conversion problem also occurs if the called procedure is a function which returns results to the caller. The problems which arise in this case are much the same as those for parameter passing.

3.6 Running a Program on an Emulator

Executing code on a "hard" machine such as a PDP10 requires loading the code into core storage and initializing the program counter register. If the machine has an operating system running on it then this process may be a very complicated one which includes making entries in various tables, thus

1. an emulator for such a machine is akin to a process, in that it is in operation the whole time the machine is available;

2. it is not possible to run a program on the machine without first making it known to the machine's private operating system.

While point 1 presents no great problems, point 2 makes it very difficult to produce a simple underlying system which allows programs to call procedures running on such a machine - the common procedure-call mechanism would seem to need an unreasonable amount of knowledge about the private operating system running on the emulator in question.

However, our original motivation for this system was the support of special purpose interpreters (rather than, for example, the study of operating systems) and we can afford to treat these interpreters differently from emulators for existing machines. In particular we propose that emulator invocations should be created and die in the same way as invocations for the procedures they run.

Note that we are not banning PDP10-like emulators from our system, we are simply saying that they cannot communicate as easily with the system as purpose-built emulators can and we are primarily interested in the latter.

3.7 The Representation of a Procedure

Let us now give a possible data structure for a procedure (Fig. 2).

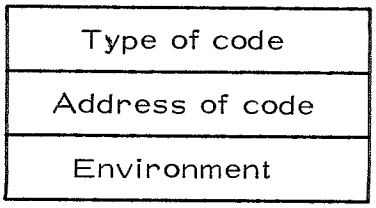


Fig. 2

This data structure corresponds to a "closure" (Landin, 8). We note that it contains the minimum of information for our purposes and a particular implementor might like to add to it, for example, the number and types of the parameters and results .

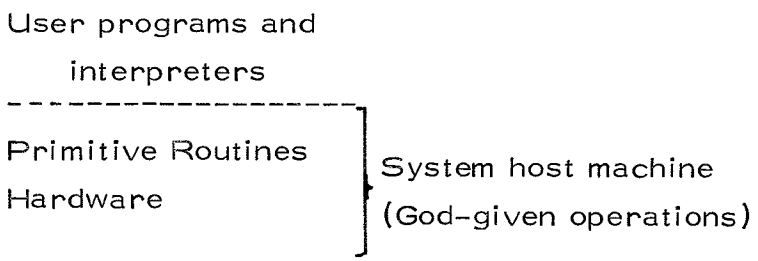
The "Type of Code" element tells the system what interpreter the procedure runs on. Its exact form is not important: it might be a string - "xcode", or an index to a system table of interpreters - "8", or a pointer to the data structure for some specific interpreter. We note in passing that "ALGOL 60 (text)" might be an acceptable code type in our system, where the execution of a procedure of this type could involve compilation as well as interpretation.

The "Address of Code" is the common format address of a vector containing the code. (How the code got there in the first place is a difficult question which we shall pass over).

The "Environment" is the means of getting at non-local storage which is referenced within the procedure body. Its form may vary from interpreter to interpreter but it is likely to be the (common format) address of some data structure (such as a display in the case of ALGOL 60).

3.8 The Procedure-call Primitive

The common procedure call operation is likely to be quite a complicated one. We note that it cannot be implemented as a procedure itself (or calling a procedure would necessitate calling the call procedure ..... ad infinitum) and so it must be a primitive operation of our system. We have christened such operations "god-given"; in practice they will be written in host machine code, but will lie outside our multi-emulator system. Thus our "system host machine" is comprised of the physical host machine and some basic routines



An interpreter running on the system host machine can call an external procedure by executing the relevant host machine instruction, but what about interpreters running at a higher level? In order to provide the generality we seek we insist that all emulators which support external procedures must have a CALL EXTERNAL instruction in the instruction set they emulate. The execution of the CALL EXTERNAL instruction causes execution of the CALL EXTERNAL instruction of the next interpreter down in the nest and so on down to the host machine.

3.9 The action taken on Procedure Call

The procedure call primitive must

1) build a system record containing control information;
2) claim local storage for the procedure, if necessary;
3) make the parameters available to the procedure;
4) make the environment available to the procedure;
5) start up the procedure's interpreter at the right code address.

The order of operations 1 - 4 may be altered somewhat, but operation 5 is the last one which the system can perform. Starting up the interpreter includes starting up any interpreters lying under it, down to the host machine where the operation will involve a jump to the starting address of the bottommost interpreter. Once this jump is made, the whole interpreter nest is running and we are no longer inside a system routine. Thus the

necessary data structures must be set up before taking this irrevocable step. This point is crucial, and we advise the reader to re-read this paragraph if he is not completely sure about it.

Starting up an interpreter could be regarded as a procedure-call of the interpreter with pointers to the parameters, environment and code of the interpreted procedure as parameters. If we implement it in this way, then our procedure call primitive is a bounded (unless the interpreter nest contains a loop) recursive operation. (We note that since the recursion is necessarily the last action in each recursive step, we can implement it iteratively.) The last recursive step is to start up the bottommost interpreter which runs on the host machine. This is a normal host machine procedure call, requiring the setting of some host machine registers, including the instruction counter.

3.10 The Representation of an Interpreter

We thus see that an interpreter in our system is simply a three parameter procedure and the data structure representing it can be the same as that for any other procedure, although it may be entered in a different system table from that for normal procedures. If a procedure Fred runs on the X-code machine, then each time Fred is called a new instance of the X-code interpreter is created in just the same way as a new invocation of Fred is created.

3.11 The Return Primitive

The procedure-return mechanism is another god-given operation, and once again each interpreter must include a RETURN EXTERNAL instruction. It is necessary to transmit the results to the caller, return the structures claimed during procedure call and to restart execution after the point of call. We note that if the invocations of each virtual machine are completely distinct, then the only information we need save on procedure entry is that pertaining to the host machine, since restoring this automatically restores the state of all the other machines as well. However, if the invocations share a common virtual store, then an emulator's CALL EXTERNAL and RETURN EXTERNAL instructions must save and restore the virtual machine state in the same way as for an internal procedure call.

3.12 Multiprogramming

Up till now we have been thinking about a single-program system. However, the ideas are sufficiently general to apply to a multiprogrammed one as well. In this case the operating system structures will be more complicated, but the representation of procedures and interpreters, and the mechanics of a procedure call should remain much the same.
We note, however, that in the system as described above only the host machine would be multiprogrammed: each process would have its own interpreter nest. Thus, if A - E are procedures running as processes, and W - Z are interpreters the system structure at a particular time might look like Fig. 3.

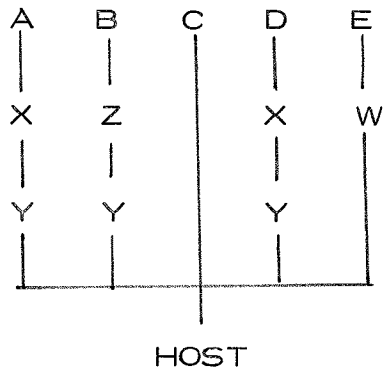


Fig. 3.

It might be desirable to allow multiprogramming of the emulators themselves, as in Fig. 4.

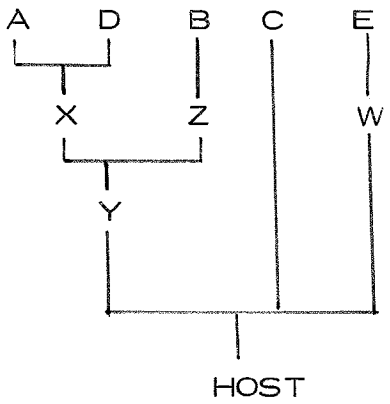


Fig. 4.

We would certainly want to do this if we were, for example, studying scheduling algorithms for various machines. However, in this case we must

1) let the system know that X can be multiprogrammed; and

2) let X know that it must run process D as well as process A

and the extra complication which arises is frighteningly large and may not be justified in view of our original aim, which was to support high-level-language-based machines.

We also note that if the system contains god-given create-process, destroy-process, signal and wait instructions, then we can implement an external procedure call of Fred () as

```
Create process  Fred (My Semaphore)
Wait (My Semaphore)
Destroy process  Fred
```

While the procedure return from Fred (caller's Semaphore) is

```
Signal (caller's Semaphore)
Wait (For some event which never happens).
```

This may not be as efficient as using special procedure call and return mechanisms, but the saving of two primitive operations may justify the extra cost.

4. SUSPENSION

It should by now be clear to the reader that we do not offer any neat, finished solutions to the problems which arise when one tries to build a multi-interpreter system. Indeed we cannot even claim to have stated all of the problems themselves. However, we hope that we have given the major ones, and that we have indicated lines of attack for their solution. Of course there is a great deal more to be done.

REFERENCES

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

2. D.B. Wortman: Language Directed Computer Design,
International Workshop on Computer Architecture,
Grenoble, 1973.

3. (Burroughs B6700)
E. I. Organick: Computer System Organization,
Academic Press 1973.

4. W. T. Wilner: Design of the Burroughs B1700,
AFIPS Fall Joint Computer Conference 1972.

5. Dec System 10, System Reference Manual,
Digital Equipment Corp., Maynard, Mass.

6. Ole Sørensen: The emulated Ocode machine for the support of BCPL,
DAIMI PB-45, Computer Science Dept., Aarhus University,
Denmark.

7. M. Richards: The BCPL Reference Manual,
Technical Memorandum No. 69/1-2, The Computer Laboratory,
Corn Exchange Street, Cambridge, England.

8. P. J. Landin: The Mechanical Evaluation of Expressions,
Computer Journal vol. 6, pp. 308-320, 1964.