BBN Report No. 2334 1 March 1972

# A MODEL AND STACK IMPLEMENTATION OF MULTIPLE ENVIRONMENTS

by

Daniel G. Bobrow

Computer Science Division
Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

Ben Wegbreit

Harvard University
Center for Research in Computing Technology
Cambridge, Massachusetts

60

This research was supported by the Advanced Research Projects Agency under ARPA Order No. 1967; Contract No. DAHC-15-71-C-0088

R

**DOCUMENT CONTROL DATA - R & D**

*(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)*

| | |
|---|---|
| 1. ORIGINATING ACTIVITY (Corporate author) Bolt Beranek and Newman Inc. 50 Moulton Street Cambridge, Massachusetts | 2a. REPORT SECURITY CLASSIFICATION Unclassified; 2b. GROUP |
| 3. REPORT TITLE A MODEL AND STACK IMPLEMENTATION OF MULTIPLE ENVIRONMENTS | |
| 4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Scientific | |
| 5. AUTHOR(S) (First name, middle initial, last name) Daniel G. Bobrow | |
| 6. REPORT DATE 1 March 1972 | 7a. TOTAL NO. OF PAGES 59; 7b. NO. OF REFS 33 |
| 8a. CONTRACT OR GRANT NO. DAHC15-71-C-0088 | 9a. ORIGINATOR'S REPORT NUMBER(S) BBN Report No. 2334 |
| b. PROJECT NO. ARPA ON 1967 | |
| c. | 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) |
| d. | |
| 10. DISTRIBUTION STATEMENT Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce for sale to the general public. | |
| 11. SUPPLEMENTARY NOTES This research was sponsored by the Advanced Research Projects Agency under ARPA Order No. 1967. | 12. SPONSORING MILITARY ACTIVITY |

13. ABSTRACT

Many control and access environment structures require that storage for a procedure activation exist at times when control is not nested within the procedure activated. This is straightforward to implement by dynamic storage allocation with linked blocks for each activation, but rather expensive in both time and space. This paper presents an implementation technique using a single stack to hold procedure activation storage which allows retention of that storage for durations not necessarily tied to control flow. The technique has the property that in the simple case, it runs identically to the usual automatic stack allocation and deallocation procedure. Applications of this technique to multi-tasking, coroutines, backtracking, label-valued variables, and functional arguments are discussed. In the initial model, a single real processor is assumed, and the implementation assumes multiple-processes coordinate by passing control explicitly to one another. A multi-processor implementation requires only a few changes to the basic technique, as described.

Unclassified
Security Classification

| 14. KEY WORDS | LINK A | | LINK B | | LINK C | |
|---|---|---|---|---|---|---|
| | ROLE | WT | ROLE | WT | ROLE | WT |
| control structures | | | | | | |
| environments | | | | | | |
| stack allocation | | | | | | |
| dendrarchy | | | | | | |
| retention | | | | | | |
| dynamic storage allocation | | | | | | |
| access environments | | | | | | |
| FUNARG problem | | | | | | |
| multi-tasking | | | | | | |
| coroutines | | | | | | |
| backtracking | | | | | | |
| label-valued variables | | | | | | |
| functional arguments | | | | | | |
| multiprocessor systems | | | | | | |
| extensible control structures | | | | | | |

DD FORM 1 NOV 65 1473

S/N 0101-807-6821

Unclassified
Security Classification

A-31409

# A MODEL AND STACK IMPLEMENTATION OF MULTIPLE ENVIRONMENTS

by

Daniel G. Bobrow

Computer Science Division
Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

Ben Wegbreit

Harvard University
Center for Research in Computing Technology
Cambridge, Massachusetts

## ABSTRACT

Many control and access environment structures require that storage for a procedure activation exist at times when control is not nested within the procedure activated. This is straightforward to implement by dynamic storage allocation with linked blocks for each activation, but rather expensive in both time and space. This paper presents an implementation technique using a single stack to hold procedure activation storage which allows retention of that storage for durations not necessarily tied to control flow. The technique has the property that in the simple case, it runs identically to the usual automatic stack allocation and deallocation procedure. Applications of this technique to multi-tasking, coroutines, backtracking, label-valued variables, and functional arguments are discussed. In the initial model, a single real processor is assumed, and the implementation assumes multiple-processes coordinate by passing control explicitly to one another. A multi-processor implementation requires only a few changes to the basic technique, as described.

TABLE OF CONTENTS

## 1. Introduction

Most of the older programming languages† have a function call/return structure that operates in a strictly last-in-first-out discipline. This, particularly when coupled with recursion, invites the use of a LIFO stack to hold the storage required by function activations[7]. Such a stack provides an elegant mechanism for control, local storage, temporary storage and argument passage. A function call entails pushing the arguments onto the stack, leaving a program continuation point for the caller on the stack, and transferring to the called function. The called function uses the next _k_ stack locations for its locals, and the remainder of the stack for temporary storage used in calculating the arguments to functions which it calls. Since stacks can be implemented directly in hardware, the mechanism is not only elegant, but efficient as well.

In several programming languages currently under design[4] or construction, this happy marriage of implementation technique and language form breaks down. If, for example, a language permits co-routines, then during execution, control will jump between several co-processes, each with its own call structure. If each environment is given its own stack then it becomes difficult or impossible to allow sharing of environments among co-processes, or a dynamically varying number of co-processes. Similarly, if a language permits a function F to return a functional result G, and if G's environment includes part of F then the storage associated with F's activation may not be deleted

†For example, FORTRAN,[16] ALGOL 60,[22] MAD,[1] LISP,[19] APL,[15] and SNOBOL.[13]

on F's exit, since part of the necessary environment of G would be prematurely destroyed. A related problem arises in multiprocessing where a language allows a function F of task T to spawn a new task T'. If the environment of F is shared with T', and if the environment of F is deleted, T' must be forcibly terminated or T' will proceed with part of its necessary environment destroyed. Similar problems arise with label-valued variables, explicit pointers into the stack, and "non-deterministic" or "backtrack" programming. All these cases arise from a common circumstance: the storage associated with function activation does not obey a LIFO discipline. It is necessary to retain storage blocks for durations not related to the order of their creation.

It is fairly straightforward to allow retention if the stack is abandoned entirely. Storage blocks are obtained by dynamic storage allocation and are returned to the free storage pool when no longer accessible, either through garbage collection or deletion with a reference count. A number of languages, including Gedanken,[25] PAL,[8] Simula[5], CPL[11], Lisp 1.5, PPL[26], Oregeno[2], and PL/I[17] employ one or more facets of this technique, though not all use the full power of dynamic block storage allocation.

However, this is an unsatisfactory solution to the problem of retention. Compared to a stack, dynamic storage allocation for function activation storage suffers a number of defects. First, it requires substantially more time to allocate and reclaim blocks. Second, it results in a substantial amount of wasted space since the storage block for each function activation must be allocated large enough to hold the maximum number of temporaries that will ever be required while control resides in that activation, yet the maximum will almost never be simultaneously reached by all activations. Third, there is wasted time in a function call, since arguments must first be held in temporaries of the calling block and then moved at the time of call to the parameter positions of the called block. Fourth, in a paging environment, dynamically stored allocation blocks tends to result in more page faults, since there is no contiguity of stack end to aid in localizing references.

This paper presents a technique for retention of function activation blocks on the stack. The technique has the property that if no retention is actually required by any portion of a program then activation storage behaves as a conventional LIFO stack; if particularly simple sorts of retention are used, the stack is as effective as the 2-stack technique which has been proposed for backtracking [23]. If more complex forms of retention are used, the technique still works correctly. In general, arbitrary retention can be achieved and unneeded activation blocks can be freed either implicitly or explicitly. Further, illegal use of an explicitly freed activation block is always detected. Section 2 of the paper presents a data structure model of control which is the basis of the implementation. Section 3 discusses implementation details, and Secion 4 discusses extensions to handle shallow binding, label-valued variables, interrupts, monitoring, cooperating sequential processes, and use of multiprocessors.

## 2 A Formal Model of Environment Structures and Control

We present an information structure model (similar in spirit to Wegner[33]) which deals with control and access contexts in a programming language; it is based on consideration of the form of run-time data structures which represent program control and variable bindings. The model is designed to help clarify some relationships of hierarchical function calls, backtracking, co-routines, and multiprocess structure. Although multiprocess structures are considered, in this section only one real processer is assumed to exist and only one process is considered active at any given time. This implies that processes must explicitly hand control from one to another. This greatly simplifies interprocess communication; Dykstra's P and V operators can be written in terms of the three control primitives defined. We call a set of processes which communicate in this way "coordinated sequential processes". In Section 4.5 we extend the implementation to true multiprocessor systems.

### 2.1 The Basic Environment Structure

In a language which has blocks and procedures, new nomenclature (named variables) can be introduced either by declarations in block heads or through named parameters to procedures. Since both define access environments, we call the body of a procedure or block a <u>uniform access module</u>. Upon entry to an <u>access module</u>, certain storage is allocated for those new named items which are defined at entry. We call this named allocated storage the <u>basic frame</u> of the module. In addition, certain additional storage for the module may be required for temporary intermediate results of computation; this additional allocated storage we call the <u>frame extension</u>. The total storage is called the <u>total frame</u> of the module, or usually just the module <u>frame</u>. We refer to the two frame pieces generically as <u>segments</u>.

A frame contains other information, in addition to named variable and temporaries. When a module is entered, the callee's frame is initialized with two pointers (perhaps implicitly); one, called ALINK, is a linked access pointer to the frame(s) which contains the higher level free variable and parameter bindings accessible within this module. The other, called CLINK, is associated with control and is a generalized return which points to the calling frame. In Algol these are called the static and dynamic links respectively. In LISP, the two pointers usually reference the same frame since bindings for variables free in a module are found by tracing up the call structure chain. (An exception is the use of functional arguments, and we illustrate that below.)

At the time of a call (entry to a lower module), the caller stores in his frame extension a continuation point for the computation. For proper value checking, an expected return value type may also be stored. Since the continuation point is stored in the caller, the generalized return is simply a pointer to the last active frame.

The size of a basic frame is fixed on module entry. It is just large enough to store the parameters and the link information. However, during one function activation, the required size of the frame extension can vary widely (with of course a computable maximum) since the amount of temporary storage used by this module before calling different lower modules is quite variable. Therefore, the allocation of these two frame segments may sometimes (advantageously) be done separately and noncontiguously. This requires a link back from the frame extension to the basic frame (denoted as BLINK below). Figure 1 summarizes the contents of a frame.

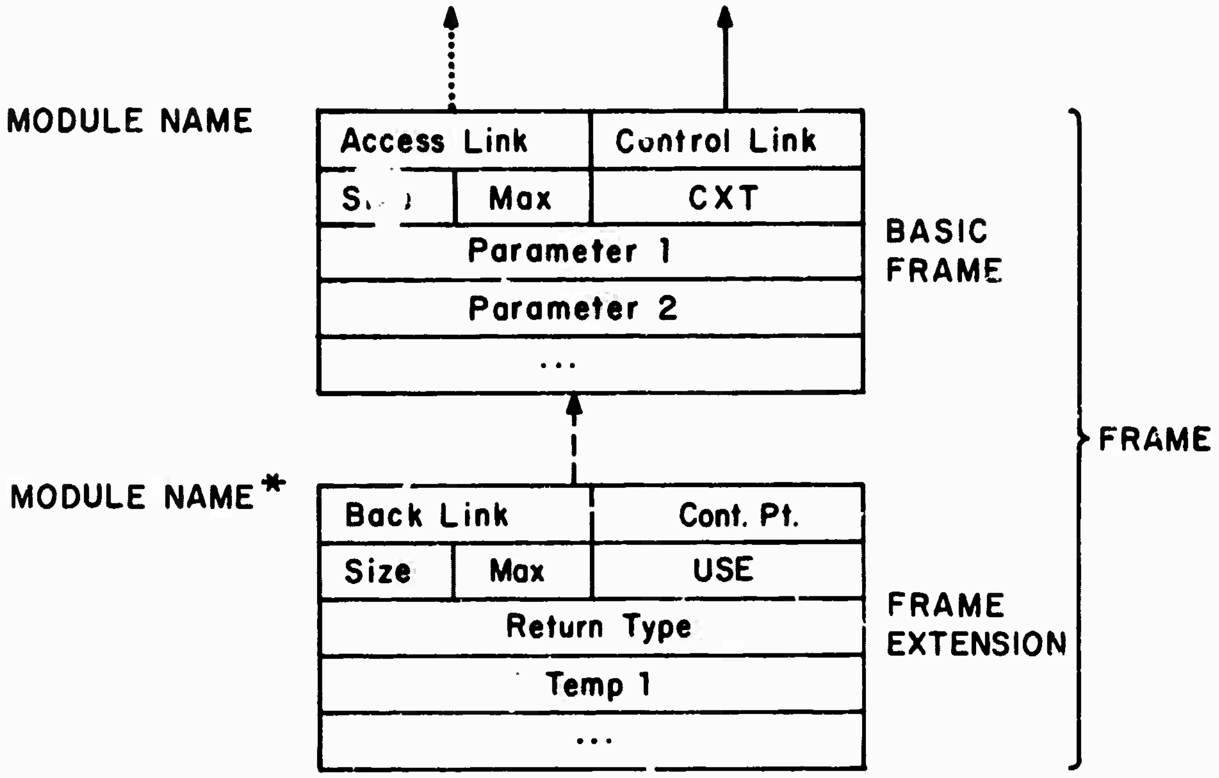


Fig. 1 General Frame Structure

Figure 2a shows a sketch of an algorithm programmed in a block structure language such as Algol 60 with contours (c.f. 18) drawn around access modules. B1 has locals N and P, P has parameter N, and B3 locals Q and L. Figure 2b is a snapshot of the environment structure after the following sequence: B1 is entered; P is called (just above $P_1$, the program continuation point after this outer call); B3 is entered; and P is called from within B3. For each access module there are two separate segments - one for the basic frame (denoted by the module name) and one for the frame extension (denoted by the module name*). Note that the sequence of access links (shown with dotted lines) goes directly from P to B1* and is different than the control chain of calls. However, each points higher (earlier) on the stack.

A point to note about an access module is that it has no knowledge of any module below it; if an appropriate value (as specified by the <u>return value type</u>) is provided, continuation in that access module can be achieved with only a pointer to the continued frame. No information stored outside this frame is necessary.

Figure 3 shows two examples in which more than one independent environment structure is maintained. In Figure 3a, two coroutines are shown which share common access and control environment A. However, note that the frame extension of A has been copied so that returns from B and Q may go to different continuation points. Since frame A is used by two processes, if either coroutine were deleted, the basic frame for A should not be deleted. Note however, that one frame extension A* could be deleted in that case, since frame extensions are never referenced directly by more than one process. In figure 3b, coroutine Q is shown calling a function [illegible] with external access chain through [illegible], but with control to return to Q.

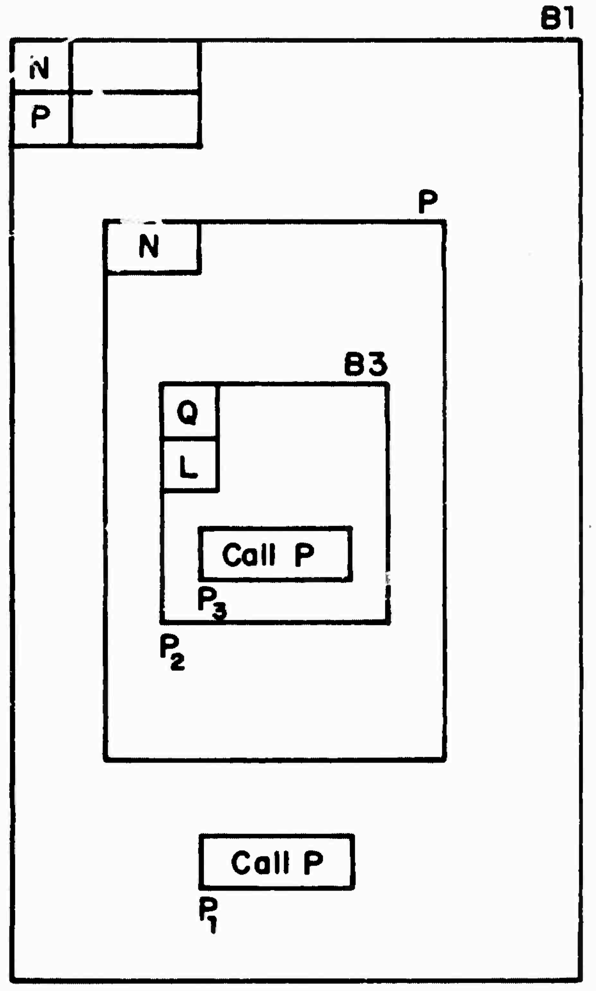


Fig. 2a (from Johnston)

Block B1 with locals N, P
Procedure P with new variable N
Block B3 with locals Q, L
Calls to P within B1 and B3

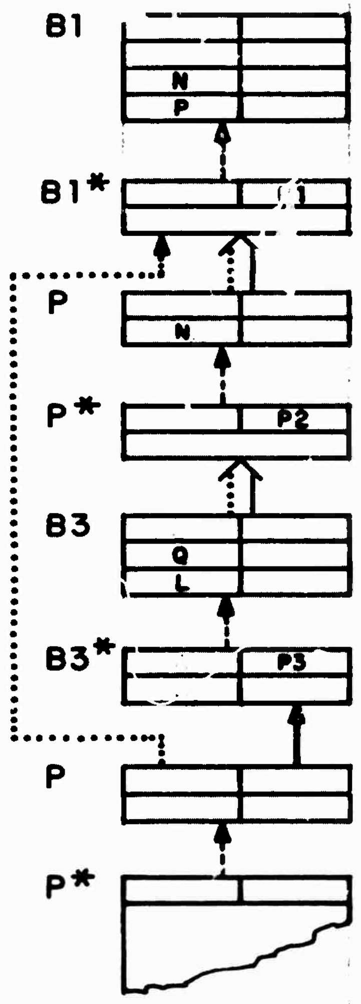


Fig. 2b Snapshot of Frame Structure Starting at B1, call to P, Enter B3, call to P.

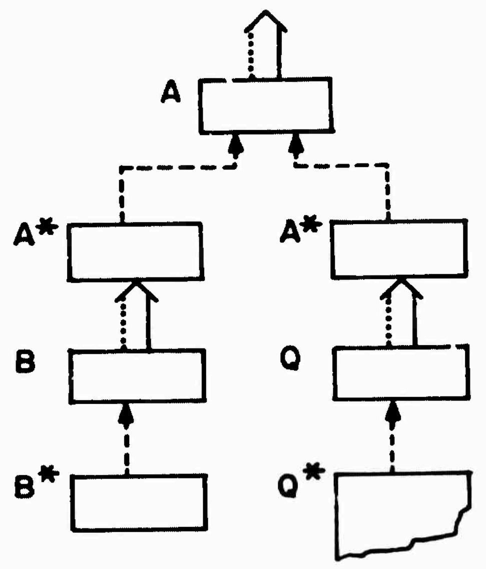


Fig. 3a Coroutines Sharing Ancestor Module A, Q is Active

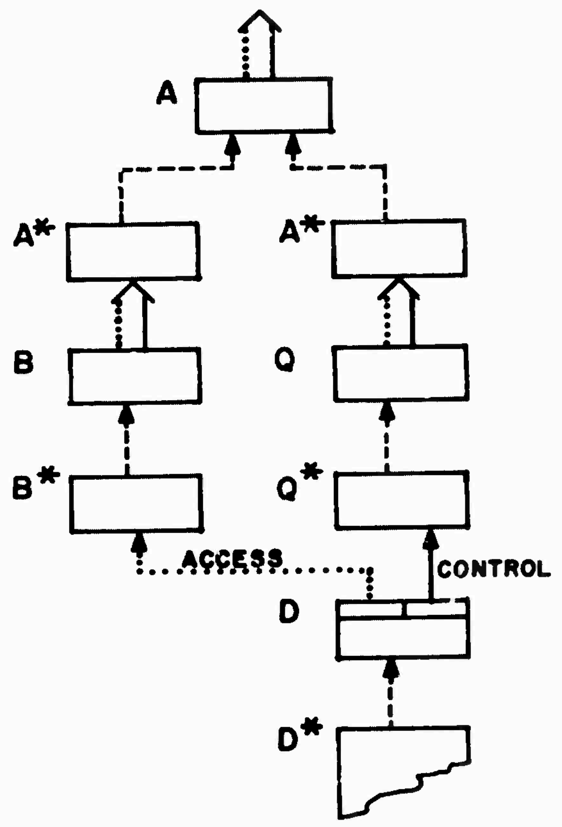


Fig. 3b Coroutine Q Evaluating Form D in Access Context of B

## 2.2 Primitive Functions for Retention

In this model for access module activation, each frame is generally released upon exit of that module. Only if a frame is still referenced is it retained. All non-chained references to a frame (and to the environment structure it heads) are made through a special data type called an _environment descriptor_. Note that heads of all environment chains but that for the currently active process are referenced from this space of environment descriptors. The three primitive functions: 1) create an environment descriptor (_ed_) for a specified frame; 2) change contents of an _ed_; 3) create a new frame with access and control chains specified by _ed_'s and execute a computation in that context. Note that none of the primitives manipulate existing frames or pointers; therefore only well formed frame chains exist (e.g. no ring structures).

| | |
|---|---|
| environ(pos,n) | creates an environment descriptor for the frame specified by _pos_. If _n_ is given and non-zero it copies the _n_ preceding frames. This allows creation of identical contexts which do not share bindings. _n_ is usually omitted. |
| setenv(olded,pos) | changes the contents of an existing environment descriptor _olded_ to point to the frame specified by _pos_. Releases storage referenced only through previous contents of _olded_. |

enveval(form,apos,cpos) initiates a computation within an environment structure; it creates a new frame, with ALINK pointing to the frame specified by apos; CLINK pointing to the frame specified by cpos; and form the code or expression to be executed or evaluated in this new environment. If the cpos argument is omitted, it is taken to be identical to apos.

A frame specification (e.g. pos; apos; and cpos) is one of the following:

1. An integer N:
   a. N=0 specifies the frame allocated on activation of the function environ, setenv, or enveval. In each case, the continuation point is set up so that a value returned to this frame (using enveval) is returned as a value of the original call to environ, setenv or enveval.
   b. N>0 specifies the frame N links down the control link chain from the N=0 frame.
   c. N<0 specifies the frame |N| links down the access link chain from the N•0 frame.
2. The distinguished constant NIL. This value specifies global-access only to be shared, and/or control-return to the system (process halt). Doing a setenv(ed,NIL) releases frame storage formerly referenced only through ed, without tying up any new storage.
3. An ed (environment descriptor). When given an ed argument created by a prior call on environ, environ creates a new descriptor with the same contents as ed; setenv copies the contents of ed into olded.

4. A list "(ed)" consisting of exactly one ed. The contents of the listed ed are used identically to that of an unlisted ed. However, after this value is used in any of the three functions, setenv(ed,NIL) is done, thus releasing the frame storage formerly referenced only through ed. This has been combined into an argument form rather than allowing the user to do a setenv explicitly because in the call to enveval the contents are needed, so it can not be done before the call; it can not be done explicitly after the enveval since control might never return to that point.

## 2.3 Non-Primitive Control Functions

To illustrate the use of these control functions, we will define some non-primitive functions which are more familiar. (We use here the syntax and semantics of a LISP-like system; although we use the LISP idiom, the conversion to other languages is straightforward.) We will define function which creates a functional object which carries its own context, and show how the language evaluator uses this object. We will then define in terms of our basic environment manipulators some non-hierarchical control functions for backtracking and coroutine calls.

We begin with an obvious extension of enveval; we can define envapply which takes as arguments a function name and list of (already evaluated) arguments for that function. Enveval requires a form and envapply simply creates the appropriate form for enveval. (Uppercase items are literal objects in LISP).

```
envapply(fn,args,aframe,cframe) =
   enveval(list(APPLY,list(QUOTE,fn),list(QUOTE,args)),
                         aframe,cframe)
```

A central notion for control structures is a pairing of a function with an environment for its evaluation. Following LISP, we call such an object a funarg. Funargs are created by the procedure function, defined

```
function(fn)=list(FUNARG,fn,environ(1))
```

That is, in our implementation, a funarg is a list of three elements: the indicator FUNARG, a function, and an environment descriptor. (The argument to environ makes it reference the frame which called function. To get an environment other than the current one, function can be evaluated within an enveval.) A funarg list, being a globally valid data structure, can be passed as an argument, returned as a result, or assigned as the value of appropriately typed variables. When the language evaluator gets a form (fcn arg1 arg2 ... argn) whose functional object fcn is a funarg, i.e. a list (FUNARG <fn-name> <ed> ), it creates a list, args, of (the values of) arg1, arg2, ..., argn, and does

```
envapply(second(fcn),args,third(fcn),1)
```

The environment in this case is used exactly like the original LISP A-list. Moses[21] has discussed the use of function in LISP for preserving binding contexts. Figure 4 illustrates the environment structure where a functional has been passed down; the function foo with variables X and L has been called; foo called mapcar(x,function(fie)) and fie has been entered. Note that along the access chain the first free L seen in fie is bound in foo, although there is a bound variable L in mapcar which occurs first in the control chain. Since frames are retained, a funarg can be returned to higher contexts and still

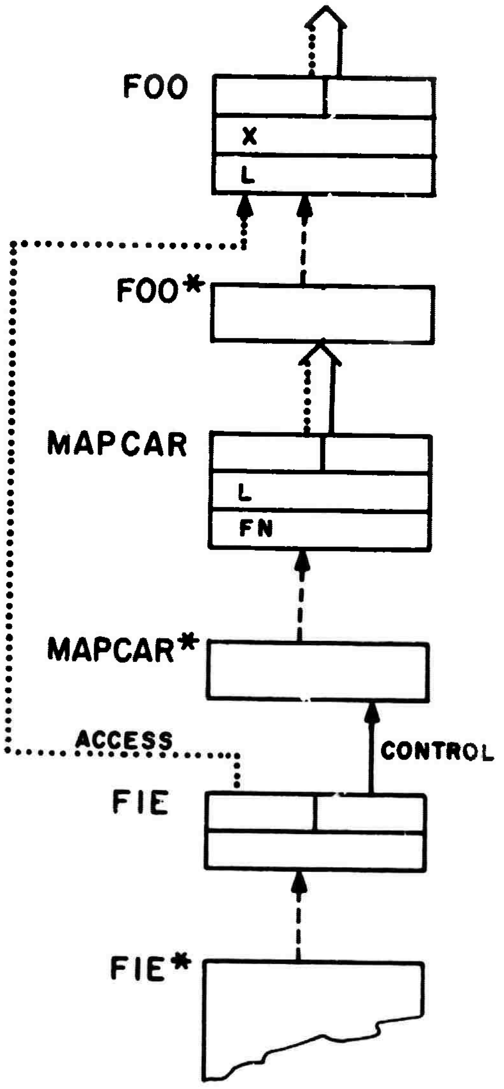


Fig. 4 Application of a Functional Argument

work. Further, as described below, funargs serve as the basis for a number of control regimes, in addition to acting as a device to save a binding environment.

Coroutines, i.e. coordinated processes which each maintain their own separate hierarchical control and access environment, are easily implemented using these primitives. A coroutine is simply a funarg used in a particular way. It is created by function and manipulated by the routines start and resume. To initiate a process represented by the funarg fp, use start:

```
    start(fp,args) = curproc←fp;
           (comment  curproc is a global variable set to
                     the current process funarg);
           envapply(second(fp),args,third(fp), third(fp))
```

Once the variable curproc is initialized, and any coroutine started, resume will transfer control between n coroutines.

```
resume(fnarg,args,backfn)=
   prog((result,flg)
       (comment  prog introduces an access module with local
                 variables result and flg.
                 backfn is the function to be called when
                 this process is resumed)
       second(curproc)←backfn
       (comment  replace old backfn for resume back here)
       result←setenv(third(curproc),0);
       (comment  result is set when a resume comes back here.
          flg will have been set when a resume comes back through
                                                          setenv.)
       if  flg then return(result);
       flg←T;
       curproc←fnarg;
       envapply(second(fnarg),args,third(fnarg),third(fnarg))
       (comment  only done first time))
```

We call a funarg used in this way a process funarg. The state of the "process" is updated by destructively modifying the list to change the continuation function, and similarly directly modifying the environment descriptor in the list. A pseudo-multiprocessing capability can be added to the system using these process funargs if each process takes responsibility for requesting additional time for processing from a supervisor by explicitly passing control. A more automatic multi-processing control regime using interrupts is discussed in section 4.4.

Backtracking is a technique by which certain environments are saved before a function return, and later restored if needed. As an example of its use, consider a function which returns one (selected) value from a set of computed values but can effectively return an alternative selection if the first selection was inadequate. That is, the current process can fail back to a previously specified failset point and then redo the computation with a new selection. A sequence of different selections can lead to a stack of failset points, and successive fails can restart at each in turn. Backtracking thus provides a way of doing a depth first-search of a tree with return to previous branch points.

We define fail and failset below. We use push(L,a) which adds a to the front of L, and pop(L) which removes one element and returns the first element of L. Failist is the stack of failset points. As defined below, fail can reverse certain changes when returning to the previous failset point by explicit direction at the point of failure. (To automatically undo certain side effects and binding changes we could define "undoable" functions which add to failist forms whose evaluation will reset appropriate cells. Fail could then eval all forms through the next ed and then call enveval.)

```
failset()=push(failist,environ(1))
  (comment  1 means environment of failset)

fail(message)=enveval(message,list(pop(failist)))

select(set,undolist)=
      if null(set) then fail(undolist)(comment reset values)
      else prog((flg)
               failset();
               if flg then return(select(set,undolist));
  (comment  flg is set if we have failed to this point, and
            then set has been popped.)

               flg←T;
               return(pop(set))
```

Floyd,[10] Hewitt,[14] Golomb, and Baumert[12] have discussed uses for backtracking in problem solving. An example of its use is the following program for placing 8 queens on a chess board such that no two can take each other. The function conflict(s,cans) (not shown) checks whether square s chosen by select for column N will fit with the previously generated answer for the first N-1 columns.

```
queens()=
   prog((n,ans,m)
      n←0;
   lp: n←n+1;
       if n>8 then return(ans);
   pl: m←select((1,2,3,4,5,6,7,8),
                (PROG()N←N-1;POP(ANS)));
       (comment Both arguments are quoted forms.
                The prog form in the select is evaluated
                only in case of a failure in select.)
       if conflict(m,ans) then fail();
       (comment  continue selection until select produces a
                 good value, or fails and resets n and ans.)
       push(ans,m);
       go(lp) )
```

Figure 5 shows the control structure saved for <u>queens</u> after it has successfully moved to the third column.

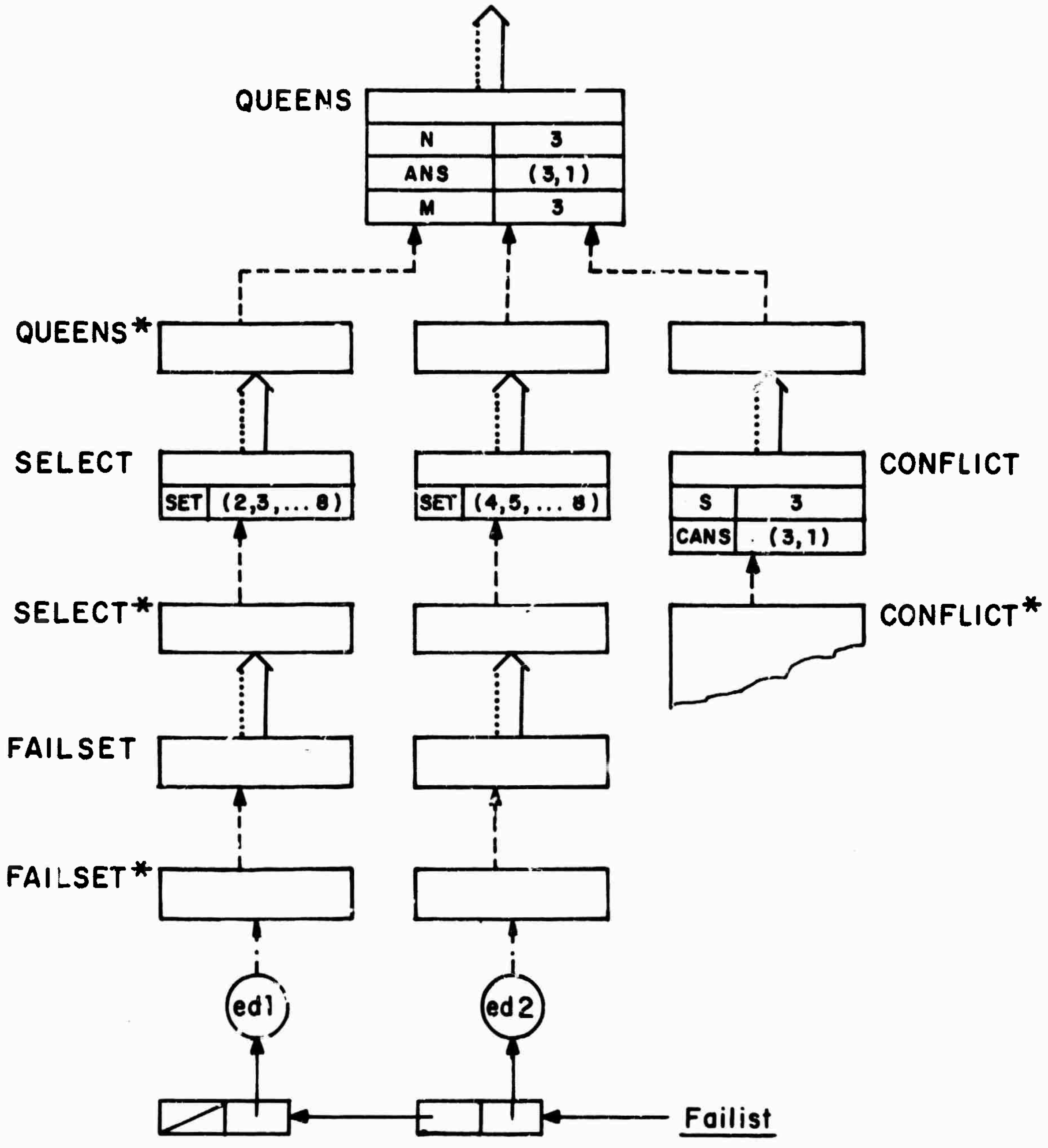


Fig. 5 Control Structure For Queens at Third Column on Chess Board

## 3. Implementation

### 3.1 Retention on the Stack

The model of section 2.1 assumes that a frame is retained so long as it is actively referenced. With a bit of bookkeeping, it is possible to determine when each frame ceases to be referenced, so that each frame can be freed by the evaluator as soon as this occurs. Further, frames can all be allocated on a single stack. This section presents the technique for so doing.

The first issue, bookkeeping of frame references, is handled by two new fields added for this purpose to each frame. A <u>basic frame</u> segment can be referenced only from its corresponding frame extensions. The CXT field in the basic frame counts the number of frame extensions for that basic frame. A <u>frame extension</u> segment can be referenced in any of three ways: 1) by the basic frame of an immediate control descendent (i.e. "callee"), 2) by the basic frame of an immediate access descendent (e.g. lower lexical range), 3) by an environment descriptor. The USE field in the frame extension counts the number of references to that frame extension.

In the case of simple LIFO control, CXT and USE are always equal to 1. <u>Environ</u> creates an environment descriptor and therefore, as part of its actions, increments by 1 the USE count of the appropriate frame extension. When the USE of a frame extension exceeds 1, the frame extension cannot be used for running in (i.e. execution) since the several users of that frame extension require the state to remain the same, but further computation in that frame extension would change the state (e.g.

destroy some temporaries and/or move the continuation point). Hence, whenever control returns to an access module where the USE count exceeds 1, a copy of the frame extension is made, USE is decremented by 1 (since there is one less user of that frame extension) and CXT is incremented by 1 (since there is one additional frame extension which references the basic frame). Figure 6 shows the structure resulting from a program in which P1 calls P2 which calls ENVIRON(1), thereby creating an environment descriptor refering to P2. When exiting any access module, the frame extension is always deleted. If the CXT in the basic frame is 1, then the basic frame is also deleted; otherwise, the basic frame remains. This, then, is the basic retention technique. We return to the details below.

The second issue, storage management with a stack, is handled as follows. On entrance to an access module, a basic frame and frame extension are pushed in contiguous locations on the end of the stack. On exit from the module, if both basic frame and frame extension are deleted, then the end of stack pointer is restored to its position on entrance. If, however, the basic frame is not deleted (CXT>1), then it remains where it is on the stack. In general, therefore, when control returns to an access module with frame extension E*, it may be that there is a basic frame immediately below E*. Suppose, for example, that procedure P0 calls P1 which calls environ(1) creating $ED_1$; P1 next calls enveval(P2(x),2,2); P2 then calls environ(1) to create $ED_2$. Figure 7a shows the stack structure and reference counts when control comes back to P2. Suppose P2 causes control to return to P1 destroying $ED_1$, (e.g. by executing enveval(TRUE,list($ED_1$)).) It is not possible to run P1* where it lies, since the basic frame of P2 blocks the stack. Hence, the evaluator makes a copy of P1*, called P1'*, at the stack end and decrements the USE count of P1*. If the new value of USE in P1* is zero, then the segment P1* is deleted. In either case, P1'* is used for further computation. Figure 7b illustrates the situation. (The dashed line is the BLINK from P1'* to P1).

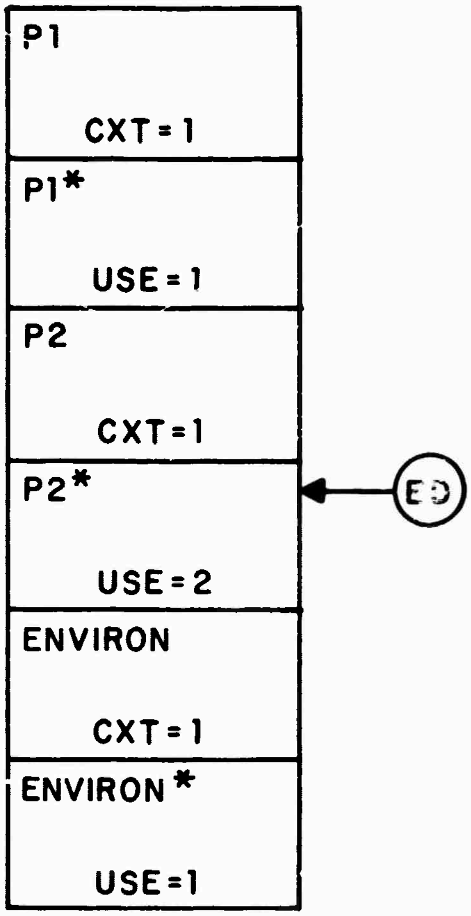


Fig. 6a Control in ENVIRON After ED Created

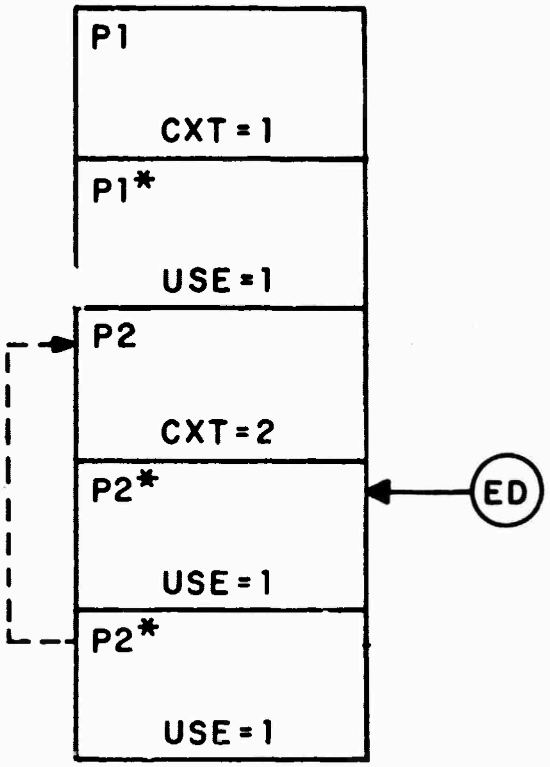


Fig. 6b Control Has Returned to P2

Fig. 6 Reference Counts for the Case P1 Calls P2 ENVIRON(1)

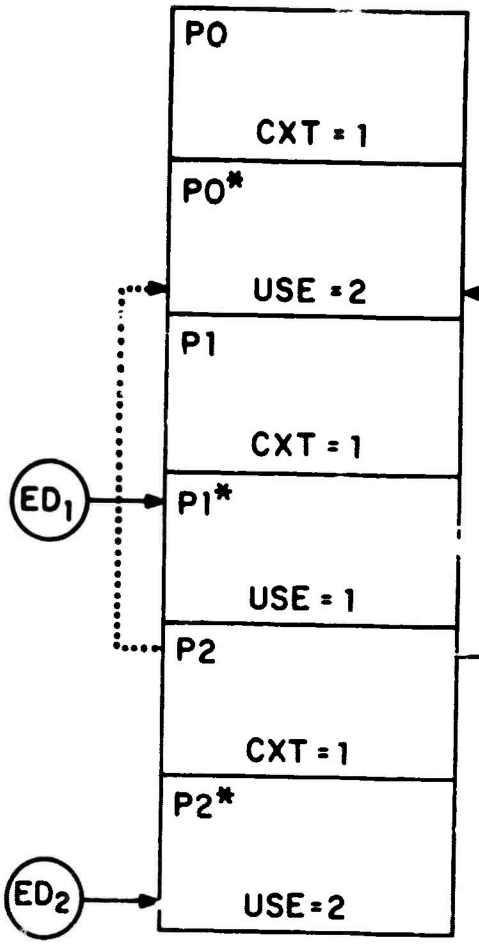


Fig. 7a Control Has Left P2 But P1 Has Not Yet Been Reentered

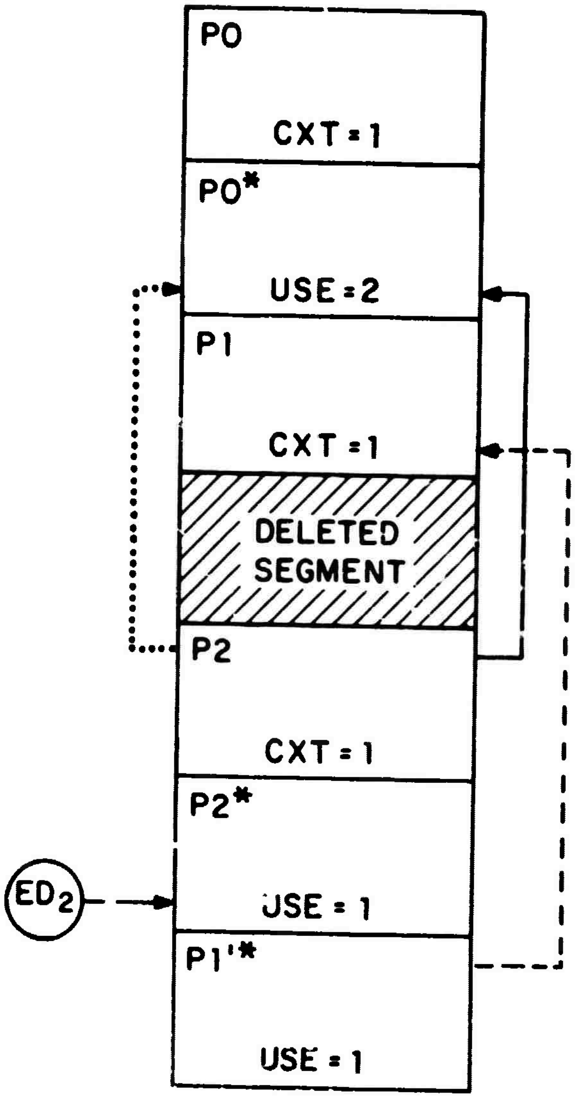


Fig. 7b Control Has Returned to Access Module P1 Using a Copy of P1*

Fig. 7 Control Returns From P2 to P1

Whenever control returns to a frame extension E* which cannot be run where it lies (due to another segment beneath and blocking it), a copy of E* is used in its place, perhaps deleting the original frame extension. Such deleted segments provide holes for the growth of the frame extensions directly above them when (if) the basic frame immediately above the hole is deleted. Hence, they serve as mini-stacks. It is the responsibility of the Delete Segment routine to appropriately record the space made available by a segment deletion so that it may be reused. We return to this issue and the issue of stack overflow in section 2.5.

With the above description of intention as an extended comment, we can now state the algorithms for using and maintaining the reference counts. Two action points during evaluation are crucial:

(1) entering an access module
(2) exiting an access module and returning to its caller

Also, the retention primitives each manipulate the reference counts

(3) environ
(4) setenv
(5) enveval

Note that these five routines cannot properly be written in the programming language. The actions used (e.g. deleting a segment) and data types employed (e.g. pointers to frames) are incompatible with the security of the evaluation mechanism, since they could be used to cause system errors. Partially to emphasize this point and partially for convenience, we switch notation. English descriptions are used where this simplest and an Algol-like syntax is used elsewhere. Liberal use is made of pointer-valued variables and the convention that if P is a pointer to a frame then P.USE, P.CXT, P.ALINK, etc. denote the fields of the basic frame and frame extension. In the case of environment descriptors, we employ a field, FPTR, which points to the frame extension for the appropriate environment.

```
Enter Access Module (F) =
begin
[1] push F and F* on stack;
[2] F.ALINK←F.CLINK←address of caller;
[3] F.CXT←F.USE←1
end


Exit Access Module (F) =
begin
[1] Delete Segment (F*);  comment  no one else can be in it, since
                          we are running in it;
[2] if F.CXT=1
        then begin  Delete Segment (F);
                    if F.CLINK≠F.ALINK then
                             Release Access Chain (F.ALINK)
             end
        else begin  F.CXT←F.CXT-1;
                    comment  next, propogate back (by incrementing
                    USE of caller) the fact that a callee still
                    exists;
                    F.CLINK.USE←F.CLINK.USE+1
                    end;
```

```
[3] let E be F.CLINK;
    comment  now return to E, the caller;
[4] if E.USE=1
     then if Sufficient Room beneath E* to run
          then Run In E*
          else begin  Copy E*; Delete Segment (E*); Run In copy  end

     else begin  E.USE←E.USE-1;
                 E.CXT←E.CXT+1;
                 Copy E*;
                 Run In copy
          end

end


Environ (POS) =
begin
[1]  Create a null environment descriptor, ED;
[2]  Environ2 (ED, POS);
[3]  Return (ED)
end
```

```
Environ2(ED,POS)

begin
[1]  let F be the frame specified by POS;
[2]  if F is the null frame then Return;
[3]  ED.FPTR←address of F*;
[4]  F.USE←F.USE+1;
[5]  if POS is a list of an environment descriptor, e.g. of
     format "(ED')", then Setenv (ED',NIL)
end


Setenv (ED,POS)=

begin
[1]  temp←ED.FPTR;
[2]  Environ2(ED,POS);
[3]  if temp≠NIL then Release Frame (temp);
[4]  Return (ED)
end


Enveval(F,APOS,CPOS) =

begin
[1]  let A be the frame specified by APOS, and C be the frame
     specified by CPOS; (if CPOS is missing, let C be A);
[2]  C.USE←C.USE+1;
     if C≠A then A.USE←A.USE+1;
```

```
[3]  let E be the frame for this call on Enveval;
     Release Frame(E);
[4]  if segment E* is not deleted in step [3] then set the
     continuation point for E* such that if control returns
     to E* with value V, then Enveval will return to its
     caller with value V;
[5]  if APOS is a list of an environment descriptor, i.e.
     "(ED)" then Setenv(ED,NIL); if CPOS is a list of an
     environment descriptor, "(ED')", then Setenv(ED',NIL);
[6]  Push a frame on the stack, with ALINK and CLINK pointing to
     A and C respectively, and evaluate form F
end


Release Frame (P) =
     comment P is always pointing to a frame extension
begin
[1]  if P.USE>1 then begin P.USE←P.USE-1; Return end:
[2]  if P.CXT>1
     then begin P.CXT←P.CXT-1;
                Delete Segment (P*);
                Return
          end ;
     comment  if neither [1] nor [2] applies then the
              entire frame is to be released ;
```

```
[3]  if P.CLINK≠P.ALINK then Release Access Chain(P.ALINK);
[4]  temp←P.CLINK;
[5]  Delete Segments (P,P*);
[6]  P←temp;
[7]  go to [1]
end


Release Access Chain (A)≡
   comment  almost identical to Release Frame (P) except this
            follows access pointers;
begin
[1]  if A.USE>1 then begin A.USE←A.USE-1; Return end
[2]  if A.CXT>1 then begin  A.CXT←A.CXT-1;
                            Delete Segment (A*);
                            Return
                     end;
[3]  temp←A.ALINK;
[4]  Delete Segments(P,P*);
[5]  A←temp;
[6]  go to [1]
end
```

As an example of the operation of these algorithms, consider the 8-queens problem. Figure 8a shows the stack immediately after environ(1) is executed in the first failset encountered. Figure 8b shows the stack when the third column of the board is being considered (situation is identical to that of Figure 5). Figure 8c is the stack configuration that would result were a conflict to occur, causing failure back to the second column. (Note that in the case of backtracking, stack storage is used and freed in strict LIFO order).

## 3.2 Storage Management, Compactification, and Garbage Collection

The above algorithms suffer from three omissions. First, they leave undefined the auxiliary routines which perform segment deletion and the test to see whether there is sufficient room beneath a module for running. Second, since a copy is made at the stack end whenever a frame extension cannot be run where it lies, the stack tends to grow ever downward. As this commonly occurs in conjunction with deleted segments occuring in the used portion of the stack, the stack may overflow although its total size does not exceed the storage actually required. Possible solutions are stack compactification to squeeze out all the holes, or keeping the holes available for running in. Third, while environment descriptors are explicitly created (by calls on environ) they may not be explicitly freed (since several pointers might reference the same environment descriptor). Hence reclaiming environment descriptors (and tracing the appropriate frames) must be carried out automatically, by garbage collection.

The basic technique for segment deletion and testing for room to run is relatively simple. Two additional fields are

Fig. 8a

Creating an Environment Descriptor

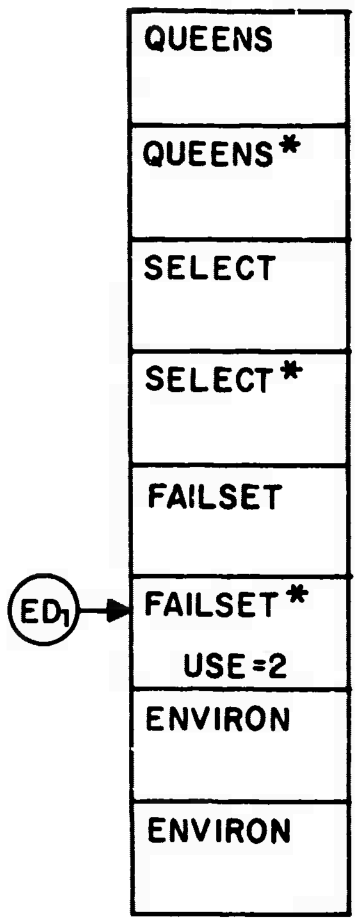


Fig. 8b

Testing For Conflict in Column 3

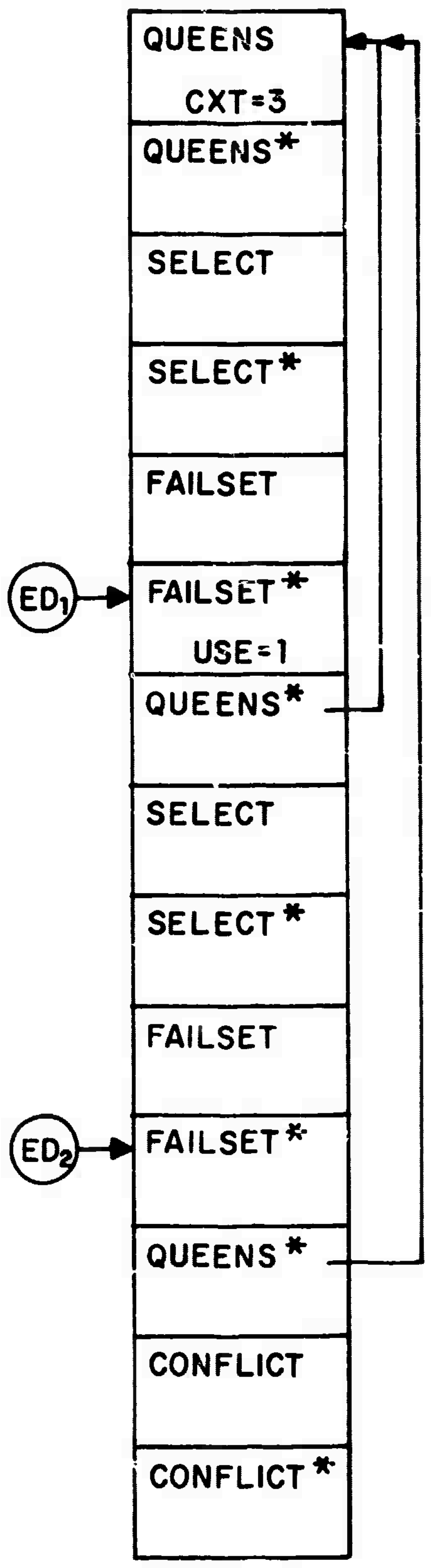


Fig. 8c

Failure Back to Column 2

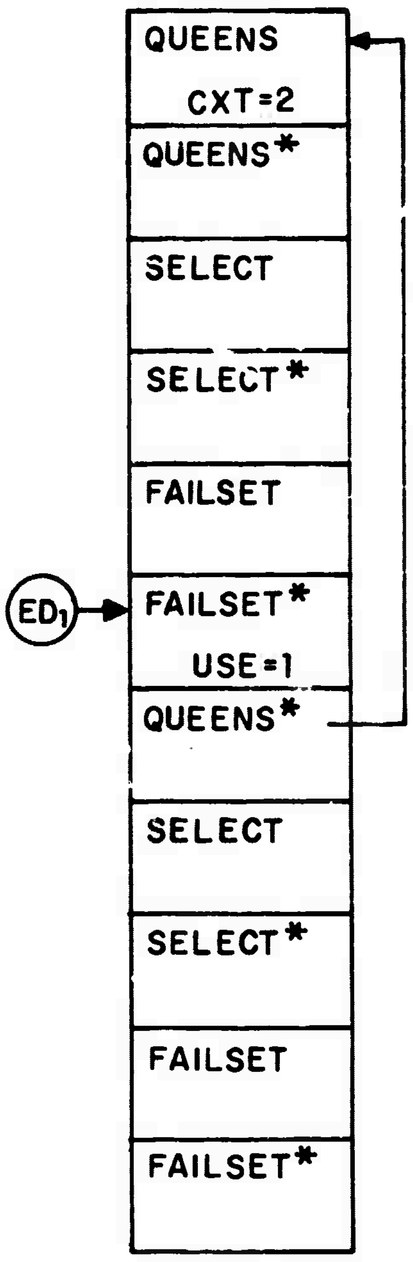


Fig. 8 Stack Configurations During Backtracking

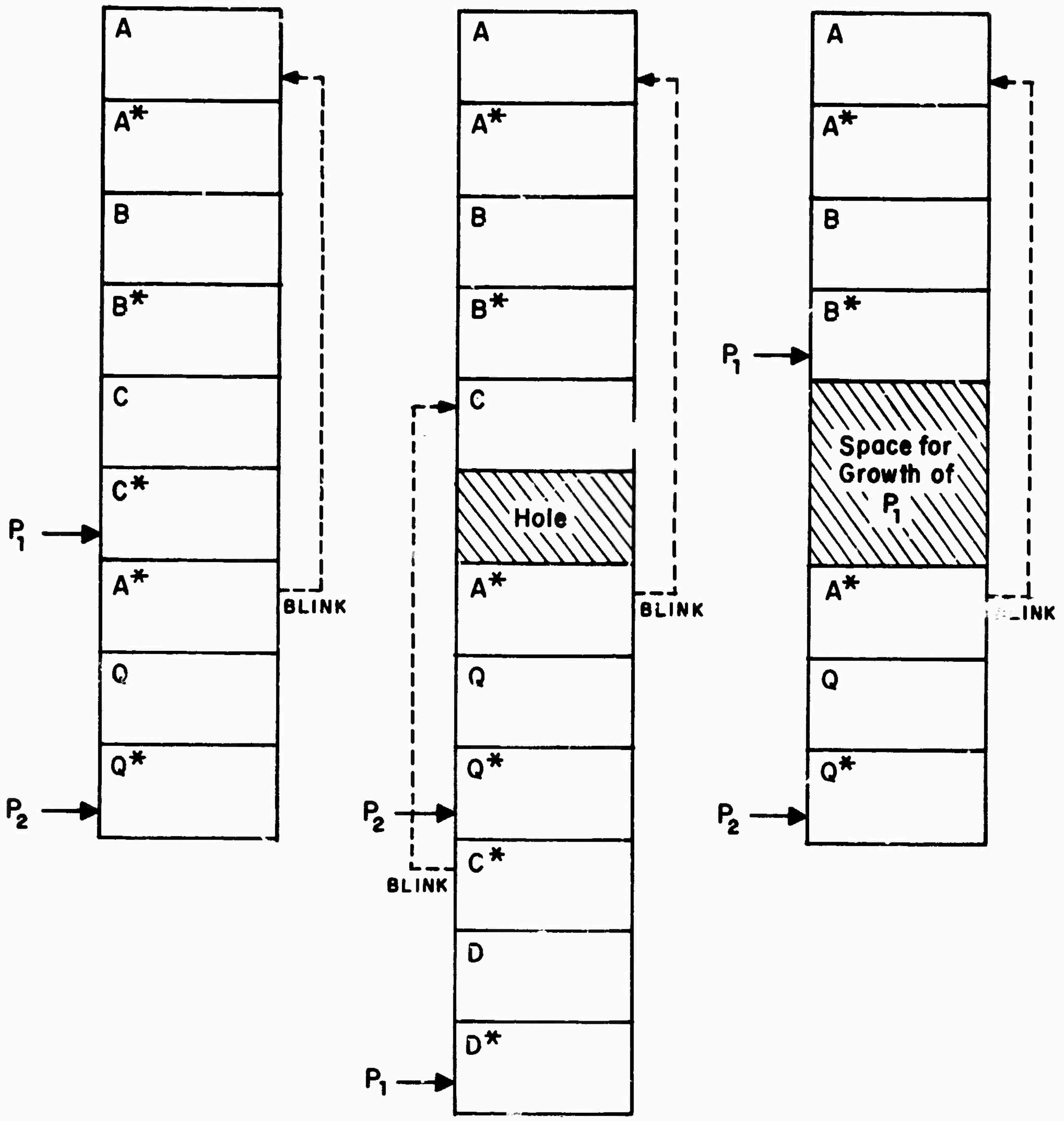
used in each segment. Each segment holds both a _size_ field which specifies its current extent (this is fixed for basic frames but varies in time for frame extensions) and a _max_ field which is the amount of free stack storage immediately below that segment. (A segment having another segment immediately below it has _max_=0).

The general situation is as follows. Computation proceeds at some point in the stack described by a _local stack descriptor_. (In general, this is not the real end of the stack but rather some hole created previously). Computation stays within the local stack region until (1) the local stack overflows, (2) a return is made from an access module G in the region to a caller F which is not in the region. In case (1), the segment which overflowed is copied elsewhere and the _max_ field of the last segment remaining in the old local stack region is set to reflect the amount of storage left in the region. In case (2), the region is being abandoned, so the region size is added to the _max_ component of the last segment above the region. When returning to F, F's _max_ is used to determine the local stack descriptor for the new stack region. There is room to run if _max_ exceeds zero. Whenever a segment is deleted, its _max_ field plus its _size_ field is added to the _max_ field of the segment immediately above it.

The effect of this technique is to break the stack up into a number of substacks (whenever multiprocessing occurs). When control returns to a module, the module is run where it lies if possible. If stack overflow occurs due to a segment S, that segment is copied to some free storage and the local stack region is temporarily abandoned. Storage for the new segment copy may be at the real end of the stack or elsewhere. We return to this point below.

As an example, consider the stack structure corresponding to the coroutine pair of Figure 3. Specifically, suppose that processes $P_1$ and $P_2$ are created by the following sequence: Module A calls B which calls C which creates a process point $P_1$ and returns to B which returns to A which calls Q which creates a process point $P_2$. The stack structure is shown in Figure 9a. Suppose $P_2$ resumes $P_1$. Since C* cannot be run where it lies, a copy is made at the stack end creating a hole above. If module C calls module D, Figure 9b results. When D returns to C the stack is simply flushed; however, when C returns to B, segments C* and C are deleted. The deletion of C provides stack space for B* to run where it lies, as shown in Figure 9c.

Two different strategies are available for handling the overflow of local stack regions. The first, the non-linearizing strategy, is the simplest and gives preferential treatment to the real end of stack. Whenever a local stack overflows, the copy is made at the real end of stack, and the remainder of the stack becomes the "current local stack". The hole at the end of the old local stack will be used only if control comes back to the corresponding frame extension. Essentially, mini-stack regions are used only by their creators, so that fragmentation is relatively common whenever co-processes occur. When an overflow occurs at the real end of stack, a stack compactification can be used to move all segments up by squeezing out all the holes. (Max fields are, of course, set to zero). This creates a single block of free storage at the real end of stack whose size is the sum of the old hole regions. Such a compactification can be carried out in a single linear sweep of the stack and requires no additional storage.



$P_2$ is active

$P_1$ is active

$P_1$ is active

Figure 9

The second, the linearizing strategy, gives no preferential treatment to the real end of stack. A pool is maintained of all the free regions on the stack. This includes the block composing the real end of stack as well as holes created by segment deletion. When control returns to a frame extension E*, it is run where it lies if the storage region beneath it is free. If not, or if the frame extension overflows its region during running, some block in the free stack region pool is chosen as the place to copy E* and continue computation. Since use of a storage block is not restricted to the process which created it, the frequency of required compactifications is substantially less than with the non-linearizing strategy. Compactification is still required, however, since fragmentation may still occur, resulting in many small useless free blocks. Further, since reuse of storage blocks is not tied to processes, there will be more interleaving of storage of different processes and more frequent overflow of local stack regions. Hence, this strategy includes linearization as part of compactification. That is, stack segments are reordered so that for each module A, some module B called by A is placed immediately below A. Techniques for such a linearization are well-known [ Minsky,[20] Bobrow[3]]. They suffer only in requiring additional storage - either in the address space or in the file system.

With either strategy there is the possibility that compactification will find few or no holes to collect. That is, stack overflow due to a large computation remains possible. With our technique this presents no problem. Computation can proceed in a new stack segment which need not be contiguous to the existing stack. Since the technique of this paper does not assume contiguity of caller and callee, non-contiguity of stack segments doesn't hurt and requires no additional mechanism.

Garbage collection of environment descriptors is a separate issue not necessarily coupled with stack compactification. All environment descriptors are allocated in the free storage region, i.e. heap. To make reclaimation simple, a region (or regions) of the heap is reserved to hold only environment descriptors. The trace and mark phase of garbage collection is standard, except that all elements of the environment descriptor block free list are marked. Hence, during the sweep phase, the only environment descriptor blocks which are picked up are those which are reclaimed by this collection. Each such environment descriptor is treated as if the program had executed setenv(ed,NIL) on this ed. That is, the associated frame is freed using the Release Frame algorithm of section 3.1. Once frame release has been carried out, the environment descriptor block is added to the existing free list of environment descriptors.

Since garbage collection of environment descriptors may free some number of stack segments, it may be useful to include such a garbage collection whenever stack compactification occurs. Alternatively, a stack compactification might be included as part of each garbage collection. Which (if either) of these is performed depends on the relative expense of garbage collection and stack compactification.

## 4. Extensions

### 4.1 Shallow Binding

The model used in section 2.1 suggests that non-local variables are accessed by searching the ALINK chain of frames. In the case of simple lexical identification for free variables (e.g. as in Algol 60) there is a well-known implementation alternative - the display of Dijkstra. If, however, dynamic identification is used for free variables (or if enveval is used to set up arbitrary environments not known at compile-time) then the display technique cannot be used. But there is a different technique for immediate access to free variables which is compatible with the general model and our implementation. With appropriate enhancements, shallow binding works correctly and efficiently.[21, 27, 31]

The basic technique of shallow binding has been used in LISP implementations for some time. The method is to associate with each atom (i.e. symbol table entry for an identifier) a special cell, the value cell, which points to the current parameter binding for that identifier. Each non-local variable in a procedure is represented by a pointer to the atom (or directly to its value cell); hence, a non-local variable can be accessed by indirecting through the value cell for that atom. Whenever a parameter binding is made or a local variable is declared, say for the variable X, the value cell is updated. The new binding for X includes a field old-adr which is set (during binding) to point to the previous parameter binding for X. When a module is exited either explicitly or implicitly (e.g. by a non-local goto) the value cell for the old value is reinstated.

With the introduction of <u>enveval</u>, the simple shallow binding strategy no longer works since application of <u>enveval</u> can change the entire set of "current" bindings. It would, of course, be possible to handle <u>enveval</u> by updating <u>all</u> variables, searching the new ALINK chain to find the new bindings. However, this is needlessly expensive.

A more sophisticated technique is to update value cells only when values are actually required. Each value cell contains an <u>indicator</u> (described below) which specifies whether or not the value is current. A variable is then accessed as follows: if the indicator specifies that the value cell is current, then it is used directly; otherwise, the access environment is searched, the proper binding found, the value cell is set to point to the current binding, and the indicator is set to reflect this.

The indicator is an <u>access chain descriptor</u> (ACD). At any point in time there is a global ACD which specifies the current access environment. An indicator in a value cell is current if and only if it is equal to the global ACD. When <u>enveval</u> is called, if the new (i.e. specified) access environment is not identical to the current environment then a new, unique, ACD is generated and becomes the global ACD. Further, if the access and control links are different, <u>and</u> the control environment is the environment of <u>enveval</u>, then the old ACD is saved (e.g. as a hidden parameter to the new frame being formed). On frame exit, there are then three possibilities: (1) if ALINK=CLINK then the normal (i.e. local) updating of parameters occurs; (2) if ALINK≠CLINK and there is an ACD which was previously saved by <u>enveval</u>, then it is restored as the global ACD; (3) otherwise, a new unique ACD is generated and becomes the new global ACD.

As to implementation, ACD's can be any unique descriptors of environments, e.g. integers or pointers to blocks allocated in the heap for this purpose. The latter has the advantage of allowing garbage collection of ACD's wnen they become unused.

## 4.2 Other References to Frames: Pointers and Label-Valued Variables

Viewed functionally, the technique of section 3.1 is merely an efficient means for insuring that frames will be retained so long as they are needed. The control primitives of section 2.2 use such frames to preserve environments for variable access and control return. There are, however, a number of other uses of frame retention for which the proposed implementation technique provides an efficient realization. Most notable are label-valued variables and explicit pointers to data objects in frames. (Reynolds uses label variables as a basis for his control structure operations in Gedanken.)[25]

Label-valued variables present a classic problem to the language implementor (e.g. Fenichel[9] ). Such a variable V may be assigned a label value belonging to a local range, for example

<u>begin</u> ... ; L: ... ; V←L; ... <u>end</u>

If the scope of V is larger than the range, then the phrase <u>goto</u> V may be encountered after the block has exited. It is then necessary to reenter the exited block. With the proposed retention technique, this presents no problem since the frame for the block can be retained so long as any label variable references a label value in the block.

Specifically, the technique is as follows. Two sorts of label values are distinguished by the implementation* - private label values and public label values. Label constants are private label values; the values of label-valued variables are public label values. A private label value may be used only in ranges lexographically contained within the module where it is defined, for example in

```
begin
...
L: ... ;
begin ... goto L ... end ;
...
end
```

Since they can only be used under safe circumstances, private label values can be implemented using standard techniques, e.g. as a pair <program address, static block number> or as a pair <program address, frame pointer>. A public label value, on the other hand, can be carried anywhere. It is implemented as a pair <program address, environment descriptor for the (least) frame containing that program address> . To insure the integrity of the public value, it is treated as a primitive data type not decomposable into its two parts. However, since the ed of such an object may want to be used in other contexts, we can extend pos to include such a possible object with the obvious interpretation.

*The distinction is an implementation i.e. compilation concept and is made only for efficiency. The programmer sees no difference and simply transacts with label values.

When an assignment of a constant label value to a label-valued variable occurs, the private label value is converted, by the evaluator, to a public one by a call on <u>environ</u> to create the appropriate environment descriptor. Subsequent assignments or parameter bindings using the public label value need not (i.e. do not) cause the creation of new environment descriptors. All label-valued variables which possess that public label value share the same environment descriptor. With this implementation, it is guaranteed that a frame is retained so long as any active label-valued variable references it. The normal garbage collection of environment descriptors frees such frames when all the relevant label-valued variables are given new values or destroyed.

Similar considerations apply to variables which can point to data objects stored in frames; i.e. problems arise if a frame is deleted while pointers to it persist. The situation does not occur in LISP since all actual data objects reside in the heap. However, in languages such as Algol 68 and PL/I, this is both possible* and grevious. (In both languages, the result is an undefined program). Again, there is a straightforward solution based on the proposed retention technique. Whenever a variable V whose scope exceeds a module R is assigned the address of a variable local to R, the (private) address is converted to a global value by pairing it (indivisibly) with an environment descriptor which references R. So long as the pointer value exists, the environment descriptor will not be garbage collected, and the frame for R and its supporting frames will be retained.

* In PL/I such a pointer value can be obtained by the built-in function ADDR.

## 4.3 Interrupts and Monitoring

### 4.3.1 Interrupts

In a practical system, provision must be made for handling the occurence of conditions which demand the interruption of an ongoing process and transfer of control by a processor to another specified process. Examples are hardware interrupts for floating point underflow/overflow, end-of-file indicator read, suspension of activity demanded by another processor, and existence of a specified monitored condition (see 4.3.2). Such interrupts are handled in our model as follows. When the interrupt occurs, the current frame is closed off. That is, the machine registers and other state information are saved in the frame extension, and the continuation point field is set to the address of a routine which will cause state restoration. Then a process funarg associated with the interrupt condition is resumed as though it were explicitly called from the closed frame, with an argument ed specifying this closed frame to be restarted.

At the point of interrupt the state of the process may be clean or unclean. An unclean state is one in which basic communication assumptions about states of pointers, queues, buffers,etc. are not true. For example, certain machine registers may contain pointers which should be traced in a garbage collection. Obviously, processes which operate when environments fail to meet appropriate assumptions must guarantee not to interact inappropriately, e.g. cause a garbage collection in the cited example. Standard techniques exist to ensure clean states when required. Software interrupts can be programmed to occur at only such points. Asynchronous hardware or real-time interrupts can perform the minimal necessary operations and induce a software interrupt for continuation at the next available time. For timely interaction, such software interrupts should be allowable at all clean points.

Each interrupt condition is identified by name. After the current frame is closed off, the <u>interrupt</u> <u>dispatch</u> <u>table</u> is searched for an entry labeled with the interrupt name. The entry has two fields: a <u>level</u> number and an <u>action</u> funarg. The <u>level</u> specifies the relative priority of the interrupt. Higher priority interrupt conditions take precedence over (and hence interrupt) lower priority levels; lower priority interrupts are queued while higher priority interrupts continue processing. When an interrupt is to be processed (i.e. its priority exceeds that of any waiting interrupt) the funarg <u>action</u> is applied (c.f section 2.3 Thomas[28] discusses a variation of this model.)

### 4.3.2 Monitoring

A useful control regime which can be built from our primitives using interrupts is that provided by a generalization of the ON CONDITION of PL/I. In essence, this allows the monitoring of a process P for attainment of a condition C. Whenever, C holds, the execution of P is interrupted and a process $P_c$ associated with the condition is executed. Since $P_c$ is programmer-defined, the effect of monitoring can be any of the following: halting execution of the job, journalizing an error but continuing, recovering from the error and continuing, normal program flow (e.g. the condition monitoring is used for dispatch logic in the main program loop).

Monitoring arbitrary conditions on contemporary machines requires a mixture of hardware and software. That is, hardware is usually used for floating point overflow, software for testing the condition $X+Y\geq 2*Z$ and sometimes hardware, sometimes software for subscript out of range. A general technique for software monitoring entails changing ordinary variables to "sensitive" ones; e.g. to monitor for the condition $X+Y \geq 2*Z$, the variables X,Y, and Z are made "sensitive" by the evaluator. (This can be implemented for example by hardware flag bits, special data types in interpreters, and special code generators in compilers). All accesses to X,Y, or Z then pass control to a general monitoring process which tests whether the variable has been changed by the access, and, if so, whether the condition being monitored now holds.

## 4.4 Coordinated Sequential Processes and Parallel Processing

It should be noted that in the model of section 2, control must be explicitly transfered from one active environment to another (by means of *enveval* or *resume*). We use the term *coordinated sequential process* to describe such a control regime. There are situations in which a problem statement is simplified by taking a quite different point of view - assuming parallel processes which synchronize only when required (e.g. by means of Dijkstra's[6] P and V operations). Using our coordinated sequential processes with interrupts, we can define such a control regime.

In our model of environment structures, there is a tree formed by the control links, a "dendrarchy" of frames. One terminal node is marked for activity by the current control bubble (the point where the language evaluator is operating). All other terminal nodes are referenced by environment descriptors or by an access link pointer of a frame in the tree. To extend the model to multiple parallel processes, k branches of the tree must be simultaneously marked active. Then the control bubble of the processor must be switched from one active node to another according to some scheduling algorithm. To meet Dijkstra's assumption of non-zero progress for each cooperating sequential process, the algorithm must guarantee each active node a minimum service.

To implement cooperating sequential processes in our model, it is simplest to think of adjoining to the set of processes a distinguished process, PS, which acts as a supervisor or monitor. This monitor schedules processes for service and maintains several privileged data structures (e.g. queues for semaphores and active processes) which are used by the parallel process manipulations functions defined below. (A somewhat similar technique is used by Prenner[24]).

The basic functions necessary to manipulate parallel active processes allow process activation, stopping, continuing, synchronization and status querying. In our single processor coordinated sequential process model these can all be _defined_ by calls (through _enveval_) to the monitor PS. Specifications for these functions are:

|  |  |
|---|---|
| process(form,apos,cpos) | this is similar to _enveval_ except that it creates a new active process P' for the evaluation of _form_, and returns to the creating process P, a _process descriptor_ (_pd_) which acts as a handle on P'. |

In this model, the _pd_ could be a pointer to a list which has been placed on a "runnable" queue in PS, and which is interpreted by PS when the scheduler in PS gives this process a time quantum. One element of the process descriptor gives the status of the process e.g. RUNNING or STOPPED. Process is defined using _environ_ (to obtain an environment descriptor used as part of the pd) and enveval (to call PS).

| | |
|---|---|
| stop(pd) | halts the execution of the process specified by pd - PS removes the process from the runnable queue. The value returned is an ed of the current environment of pd. |
| continue(pd) | returns pd to the runnable queues. |
| status(pd) | value is an indication of status of pd. |
| obtain(semaphore) | this Dijkstra P operator transfers control to PS (by enveval) which determines if a resource is available (i.e. semaphore count positive). PS either (1) hands control back to P1 (with enveval) having decremented the semaphore count, or (2) enters P1 on that semaphore's queue in PS's environment. |
| release(semaphore) | this Dijkstra V operator increments the semaphore count, and if it goes positive, it moves one process from the semaphore queue (if any exist) onto the runnable queue. It then hands control back to the calling process. |

We emphasize that these six functions can be defined in terms of the control primitive of section 2.2 coupled with use of the interrupt system.

Scheduling of runnable processes could be done by having each process (by agreement) ask for a time resource at appropriate intervals. In this scheduling model, control never leaves a process without its knowledge, and the monitor simply acts as a bookkeeping mechanism. Alternatively, ordinary time-sharing among processes on a time quantum basis could be implemented through the interrupt mechanism of 4.3. Timer interrupts could be handled by PS after the frame of the interrupted process had been closed off. The ed of the interrupted process is sufficient to restart it, and can be saved on the runnable queue within a process descriptor. Because timer interrupts are asynchronous with other processing, in such a simulated multiprocessor system, evaluation of forms in the dynamic environment of another running process cannot be done consistently; the ed obtained from stopping a process provides a consistent environment. Because of this interrupt asynchrony, in order to ensure system integrity, queue and semaphore management in PS must be uninterruptible e.g. at the highest priority level.

Having augmented our simple coordinated sequential process system with a multi-process supervisor, a variety of additional control structures may be readily created. As an example, we consider multiple parallel returns - the ability to return from a single call on a module G several different times with several (different) values. A slight generalization is to allow G to give multiple returns, perhaps to different modules higher on its control chain. For G to return from the current position to a frame fr with value given by val and still continue to run, P simply calls process(val,fr,fr). Then the current G and the new process proceed in quasi parallel.

## 4.5 Extension of Stack Mechanism For Multiple Processors

Section 4.4 describes a set of functions for handling multi-processing based on the environment primitives of section 2.3, and the interrupt facility of section 4.3. However, only one active processor was assumed. Somewhat surprisingly, the implementation technique described in section 3 still works for more than one active processor with only a few modifications in the basic technique, i.e. it implements a dendrarchy in a multiprocessor configuration.

We believe the functions for manipulation of multiple processes described in section 4.4 are a good basis set. To assure system integrity, process descriptors must be made primitive, i.e. not modifiable except through the routines described, and therefore those six functions must be built in. That is, the functions of section 4.4 and the data type process descriptor become primitives. However, for the purpose of this section, the details of process manipulation are of secondary concern; other semantic bases for multiprocessing would do as well (e.g. Prenner[24], Thomas[28].) In this section we depend only on some general underlying structures. What is of concern here is that the stack retention mechanism is still applicable under a multiprocessor regime.

Regardless of details, the general situation presents some m physical processors and k processes to be run. The process descriptors provide a handle on (i.e. "names" for) the processes. Assuming k>m, the m processors multiplex themselves over the k processes according to some scheduling algorithm (primitives to program the scheduler are not discussed here). The processes waiting for processors are kept on a queue; a processor takes a process from the queue, runs it, returns it to the queue, and repeats the cycle. We assume that processes interlock themselves (e.g. by a test-and-set busy wait loop) so that no process is ever run simultaneously by more than one processor.

Given this situation, the implementation technique of section 3 requires two sorts of augments: (1) use of critical resources must be properly synchronized, (2) appropriate processor-to-processor interrupts must be included in the system. At any point in time, each processor is running some process, using a <u>local stack segment</u>. These local stack segments are disjoint. Since at most one processor is running a process at one time, each frame extension that is actively running has a unique processor owning it. However, a basic frame or a non-running frame extension may be used by many processors; e.g. two processors can simultaneously exit the same basic frame. Hence, the CXT, USE, and <u>max</u> fields are always locked (test and set) by each processor before access and unlocked afterward.* With this processor-processor exclusion, it is guaranteed that (1) no segment will be improperly deleted, and (2) a frame extension will never be simultaneously run by more than one process.

Since the local stack segments are disjoint, there is no problem on module entrance, so long as frames can be accomodated in the segment. When a local stack segment overflows, the processor must obtain a new stack segment for its exclusive use. If there is a free segment pool (as in the linearizing technique of section 3.2), the pool is locked, a segment is obtained, and the pool is unlocked. If the pool is empty or not used (as in the non-linearizing technique of section 3.2), then the processor P1 in need of stack space calls a storage allocator which might provide a new block from the heap. Alternatively, if space is

*A process which attempts to lock a resource and finds the resource already locked goes into a busy wait loop repeatedly trying to lock it (or perhaps reschedules itself for another activity).

available in a stack segment of another processor, say P2, the allocator can obtain a portion of that space. It interrupts P2, and the interrupt routine for P2 transfers part of P2's local stack storage to P1 and changes its local stack descriptor to reflect the transfer. Thus the multiprocessor implementation still requires only one global pool of stack storage which can be dynamically allocated and reallocated among the several processors.

## 5. Conclusion

In providing linguistic facilities more complex than hierarchical control, the key problems are (1) finding a model that clearly exhibits the relation between processes, access modules, and their environment and (2) developing techniques for implementing this model with acceptable efficiency. This paper has presented a solution to both problems. The model of section 2.1 is applicable to languages as diverse as LISP, APL and PL/I and can be used for the essential aspects of control and access in each. The control primitives introduced section 2.2 provide a small basis on which one can define almost all known regimes of control. The implementation presented in section 3 is perfectly general, yet for several sub-cases (e.g. simple recursion, simple backtracking) is as efficient as each of the best known special techniques. Further, the model and technique are robust, in that they can be extended to a number of other applications and situations.

## 6. Acknowledgements

This work was supported in part by the Advanced Research Projects Agency under Contracts DAHC 15-71-C0088 and F19628-68-C-0379 and by the U.S. Air Force Electronics Systems Division under Contract F19628-71-C-0173. Daniel Bobrow is now at Xerox Palo Alto Research Center.

The authors would like to thank Madeline Morin for her patience and fortitude through endless revisions, large and small.

## REFERENCES

1. Arden, B., Galler, B., and Graham, R. The Michigan Algorithm Decoder, University of Michigan Press, December 1964.

2. Berry, D. "Introduction to Oregano", in Tou and Wegner,[29] pp. 171-190.

3. Bobrow, D.G. "Storage Management in Lisp", in Bobrow (Ed.), Symbol Manipulation Languages and Techniques, North-Holland, Amsterdam, 1968.

4. Bobrow, D.G., "Requirements for Advanced Programming Systems for List Processing." CACM, Vol. 15, No. 6 June 1972.

5. Dahl, O. and Nygaard, K. "SIMULA - An Algol-based Simulation Language, CACM, Vol. 9, No. 9 (Sept. 1966), pp. 671-678.

6. Dijkstra, E.W."Co-operating Sequential Processes," in Genuys (Ed.), Programming Languages, Academic Press, 1967.

7. Dijkstra, E.W. "Recursive Programming", Numerische Mathematik 2, (1960), 312.318. Also in Programming Systems and Languages. S. Rosen (Ed.), McGraw-Hill, New York, 1967.

8. Evans, A. "PAL - A Language for Teaching Programming Linguistics", Proc 23rd Nat. Conf., 1968, Brandon Systems Press, Princeton, N.J., pp. 395-403.

9. Fenichel, R. "On Implementation of Label Variables", CACM, Vol. 14, No. 5, (May 1971), pp. 349-350.

10. Floyd, R.W. "Non-deterministic Algorithms", J. ACM, 14, (October 1967), 636-644.

11. Garwick, J. "GPL, a Truly General Purpose Language", CACM, Vol. 11, No. 9, September 1968.

12. Golomb, S.W. and Baumert, L.D. "Backtrack Programming", J. ACM 12, (October 1965), 516-524.

13. Griswold, R.C., et alis. *The SNOBOL4 Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1968).

14. Hewitt, C. "PLANNER: A Language For Manipulating Models and Proving Theorems in a Robot", *Proc. International Joint Conference on Artificial Intelligence*, Washington, D.C., May 1969.

15. IBM, *APL/360 User's Manual*, GH20-0683.

16. IBM System/360, *FORTRAN IV Language*, Form C28-6515-4, IBM (1966).

17. IBM System/360, *PL/I Language Reference Manual*, Form C28-8201-2, IBM (1969).

18. Johnston, J.B. "The Contour Model of Block Structured Processes", in Tou and Wegner,[29] pp. 55-82.

19. McCarthy, J., et al. *Lisp 1.5 Programmer's Manual*, The M.I.T. Press, Cambridge, Massachusetts (1962).

20. Minsky, M. "A Lisp Garbage Collector Using Serial Secondary Storage", M.I.T. Project MAC, Memorandum MAC-M-129, December 1963.

21. Moses, J. "The Function of FUNCTION in LISP", *SIGSAM Bulletin* (July, 1970), 13-27.

22. Naur, P. (Ed.). "Revised Report on the Algorithmic Language ALGOL 60", *Comm. ACM*, Vol. 6, No. 1 (January, 1963), 1-17.

23. Prenner, C., Spitzen, J. and Wegbreit, B. "An Implementation of Backtracking for Programming Languages", submitted for publication, ACM-72.

24. Prenner, C. *Multi-path Control Structures for Programming Languages*, Ph.D. Thesis, Harvard University, May 1972 (forthcoming).

25. Reynolds, J. "GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept", *CACM*, Vol. 13, No. 5, (May 1970) pp. 308-319.

26. Standish, T., "PPL Implementation," Harvard University, (Personal Communication) 1971.

27. Sussman, G.J. "Muddle Language Implementation", M.I.T. A.I. Laboratory, (Personal Communication) 1971.

28. Thomas, R.H. A Model For Process Representation and Synthesis, Ph.D. Thesis, Project MAC Report TR-87, M.I.T. 1971.

29. Tou, J. and Wegner, P. (Eds.). Sigplan Notices - Proc. Symposium on Data Structures in Programming Languages, Vol. 6, No. 2, (February 1971).

30. van Wijngaarden, A. (Ed.). Report on the Algorithmic Language ALGOL 68, MR 101, Mathematisch Centrum, Amsterdam (February 1969).

31. Wegbreit, B. Studies in Extensible Programming Languages, Ph.D. Thesis, Harvard University, May 1970.

32. Wegbreit, B. "The ECL Programming System", Proc. AFIPS 1971 FJCC, Vol. 39, AFIPS Press, Montvale, N.J., pp. 253-262.

33. Wegner, P. "Data Structure Models for Programming Languages", in Tou and Wegner[29] pp. 55-82.