A MODEL FOR CONTROL STRUCTURES FOR ARTIFICIAL INTELLIGENCE PROGRAMMING LANGUAGES

by

Daniel G. Bobrow

Computer Science Division
Xerox Palo Alto Research Center
Palo Alto, California 94304

Ben Wegbreit

Harvard University
Center for Research in Computing Technology
Cambridge, Massachusetts 02138

Abstract

Newer programming languages for artificial intelligence extend the class of available control regimes beyond simple hierarchical control. In so doing, a key issue is using a model that clearly exhibits the relation between modules, processes, access environments, and control environments. This paper presents a model which is applicable to diverse languages and presents a set of control primitives which provide a concise basis on which one can define almost all known regimes of control.

1. Introduction

Newer programming languages! for artificial intelligence (e.g., PLANNER9^ CONNIVER,[18] BBN-LISP.ly QA4.1′*1* extend the nature of control regimes available to the user. In this paper, we present an information structure model20 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. We present the model and its small set of primitive operations, then define several control regimes in terms of the primitives, and then consider extensions to handle cooperating sequential processes.

2. 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 uniform access module. Upon entry to an access module, certain storage is allocated for those new named items which are defined at entry. We call this named allocated storage the basic frame 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 frame extension. The total storage is called the total frame for the module, or usually just the module frame.

A" frame contains other information, in addition to named variables and temporaries. It is often useful to reference a frame by symbolic nomenclature. For this purpose, each frame has a framename (usually the procedure name). When a module is entered, its frame extension is initialized with two pointers (perhaps implicitly); one, called A LINK, 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 LISpH 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. 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 associated information. However, during one function activation, the required size of the frame extension can vary widely (with 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 n on contiguously. This requires a link (BLINK) from the frame extension to the basic frame which contains the bindings.

When a frame is exited, either by a normal exit or by a non-local goto which skips the frame (e. g. , an error condition), it is often useful to perform clean-up action for the frame. Examples include: close files opened by the frame which are no longer needed, restore the state of more global structures which have been temporarily modified by the frame, etc. Terminal action for a frame is carried out by executing an exit function for the frame, passing it as argument the nominal value which the frame is returning as its result; the value returned by the exit function is the actual value of the frame. The variable values and the exit function are the only components of the frame which can be updated by the user; all the others are fixed at the time of frame allocation. Figure 1 summarizes the contents of the frame.

Figure 2a shows a sketch of an algorithm programmed in a block structure language such as Algol 60 with contoursIO drawn around access modules. Bl 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: Bl is entered; P is called (just above P I , the program continuation point after this outer call); B3 is entered; and F 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 B I* and is different from 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 (i.e., one whose type agrees with the stored return type) 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. Note that the frame extension of A has been copied so that returns from B and Q may go to different continuation points. This is a key point in the model; whenever a frame extension is required for conflicting purposes, a copy is made. Since frame A is used by two processes, if either coroutine were deleted, the basic frame for A should not be deleted. However, one frame extension A* could be deleted in that case, since frame extensions are never referenced directly by more than one process. Since the basic frame A is shared, either process can update the variable bindings in it; such changes are seen both by B and *Q,* In Figure 3b, coroutine Q is shown calling a function D with external access chain through B, but with control to return to Q.

3. Primitive Functions

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 protected data type called an environment descriptor, abbreviated ed. The heads of all environment chains are referenced only from this space of descriptors. (The one exception is the implicit ed for the currently active process.) The primitive functions create an ed for a specified frame and update the contents of an ed; create a new frame with specified contents, and allow execution of a computation in that context; and access and update the exit function for a frame. Note that none of the primitives manipulate the links of existing frames; therefore, only well-formed frame chains exist (i.e., no ring structures).

1) environ(pos) — creates an environment descriptor for the frame specified by pos.

2) setenv(olded, pos) -- changes the contents of an existing environment descriptor olded to point to the frame specified by pos. As a side effect, it releases storage referenced only through previous contents of olded.

3) mkframe(epos,apos, epos,bpos,bcopflg) -- creates a new frame and returns an ed for that frame. The frame extension is copied from the frame specified by epos, and the ALINK and CLINK are specified by apos and epos, respectively. The BLINK points to the basic frame specified by bpos, or to a copy of the basic frame if bcopflg=TRUE. In use, arguments may be omitted; bcopflg is defaulted to FALSE; apos, bpos and epos are defaulted to the corresponding fields of the frame specified by epos. Thus mkframe(epos) creates a new frame extension identical to that specified by epos.

4) enveval(forrA,apos,cpos) — creates a new frame and initiates a computation with this environment structure. ALINK and CLINK point to frames specified by apos and epos, respectively; and form specifies the code to be executed, or the expression to be evaluated in this new environment. If apos or cpos are omitted, they are defaulted to the ALINK or CLINK of this invocation of enveval. Thus, enveval(form) is the usual call to an interpreter, and has the same effect as if the value of form had appeared in place of the simple call to enveval.

5) setexfn(pos,fn) — places a pointer to a user defined function in the exitfn field of the frame pos. If the system is using the exitfn, this will create a new function which is the composition of the user function (applied first) and the system function. On frame exit, the exitfn will be called with one argument, the value returned by the frame code; the value returned by fn will be the actual value returned to the frame specified by CLINK.

6) getexfn(pos) — gets the user set function stored in exitfn of frame pos. Returns NIL if none has been explicitly stored there.

7) framenm(pos) -- returns the framename of frame pos.

A frame specification (i.e. , pos, apos, bpos, epos, epos above} is one of the following:

1. An integer N:
   a. N=0 specifies the frame allocated on activation of the function environ, setenv, etc. In the case of environ, setenv and mkframe, 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 mkframe.
   b. N>0 specifies the frame N links down the control link chain from the N=0 frame.
   c. N<0 specifies the frame INI links down the access link chain from the N=0 frame,
2. A list of two elements (F,N) where F is a framename and N is an integer. This gives the Nth frame with name F, where a positive (negative) value for N specifies the control (access) chain environment.
3. The distinguished constant NIL. As an access-link specification, NIL specifies that only global values are to be used free. A process which returns along a NIL control-link will halt. Doing a setenv(ed,NIL) releases frame storage formerly referenced only through ed, without tying up any new storage.
4. 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.
5. 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 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 cannot be done before the call; it cannot be done explicitly after the enveval since control might never return to that point.

4. Non-Primitive Control Functions

To illustrate the use of these primitive control functions, we explain a number of control regimes which differ from the usual nested function call-return hierarchical structure, and define their control structure routines in terms of the primitives. We include stack jumps, function closure, and several multiprocessing disciplines. In programming examples, we use the syntax and semantics of a LISP-like system.

In an ordinary hierarchical control structure

system, if module F calls G, G calls H, and H calls J, it is impossible for J to return to F without going back through G and H. Consider some program in which a search is implemented as a series of such nested function calls. Suppose J discovered that the call to G was inappropriate and wanted to return to F with such a message. In a hierarchical control structure, H and G would both have to be prepared to pass such a message back. However, in general, the function J should not have to know how to force intermediaries; it should be able to pass control directly to the relevant module. Two functions may be defined to allow such jumpbacks. (These are implemented in BBN-LISP;19 experience has shown them to be quite useful.) The first function, retfrom(form,pos), evaluates form in the current context, and returns its value from the frame specified by pos to that frame's caller; in the above example, this returns a value to G's caller, i.e., P, The second function, retevaKform, pos), evaluates form in the context of the caller of pos and returns the "value of the form to that caller. These are easily defined in terms of enveval:

```
retfrom(form,pos) = enveval{form,2,pos)
retevalform, pos)  = envevalform, pos, pos)
```

(The second argument to retfrom establishes that the current environment is to be used for the evaluation of form.)

As another example of the use of retfrom, consider an implementation of the LISP error protection mechanism. The programmer "wraps a form in errorset", i.e., errorset(form) which is defined as cons(eval(form),NIL). This "wrapping" indicates to the system the programmer's intent that any errors which arise in the evaluation of form are to be handled by the function containing the errorset. Since the value of errorset in the non-error case is always a list consisting of one element (the value of form), an error can be indicated by forcing errorset to return any non-list item. Hence, the system function error can be defined as retfrom(NIL,(ERRORSET 1)) where uppercase items are literal objects in LISP. This jumps back over all intermediary calls to return NIL as the value of the most recent occurrence of errorset in the hierarchical calling sequence.

In the following, we employ envapply which takes as arguments a function name and list of (already evaluated) arguments for that function. Envapply simply creates the appropriate form for enveval.

```
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(2))
```

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.) 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 (fen arg1 arg2 ... argn) whose functional object fen is a funarg, i. e. , a list (FUNARG fn-name ed), it creates a list, args, of (the values of) argl, 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 12 and Weizenbaum[25] have discussed the use of function 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 work. (Burge[3] gives examples of the use of funargs passed up as values.)

In the above description, the environment pointer is used only to save the access environment. In fact, however, the pointer records the state of a process at the instant of some call, having both access and control environments. Hence, such an environment pointer serves as part of a process handle. It is convenient to additionally specify an action to take when the process is restarted and some information to be passed to that process from the one restarting it. The funarg can be reinterpreted to provide these features. The function component specifies the first module to be run in a restarted process, and the arguments (evaluated in the caller) provided to that function can be used to pass information. Hence, a funarg can be used as a complete process handle. It proves convenient for a running process to be able to reference its own process handle. To make this simple, we adopt the convention that the global variable curproc is kept updated to the current running process.

With this introduction, we now define the routines start and resume, which allow control to pass among a set of coordinated sequential processes, i.e. , coroutines, in which each maintains its own control and access environment (with perhaps some sharing). A coroutine system consists of n coroutines each of which has a funarg handle on those other coroutines to which it may transfer control. To initiate a process represented by the funarg fp, use start (we use brackets below to delimit comments):

```
start(fp.args) = curproc — fp;
      [ 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. The control point saved is just outside the resume, and the user specifies a function (backfn) to be called when control returns, i.e., the process is resumed. This function is destructively inserted in the funarg list. The args to this function are specified by the coroutine transferring back to this point.

```
resume(fnarg,args,backfn) =
    second( curproc) — backfn;
        [save the specified backfn for a subsequent
         resume back here]
    setenv(third(curproc), 2);
        [environment saved is the caller of resume]
    curproc — fnarg;
        [set up curproc for the coroutine to be
         activated]
    envapply(second(fnarg),args,third(fnarg),
             third (fnarg))
        [activate the specified coroutine by applying
         its backfn to args]
```

We call a funarg used in this way a process funarg. The state of a "process" is updated by destructively modifying a list to change its continuation function, and similarly directly modifying its 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 or by explicitly passing control as in CONNIVER,[18] A more automatic multiprocessing control regime using interrupts is discussed later.

Backtracking is a technique by which certain environments are saved before a function return, and later restored if needed. Control is restored in a strictly last saved, first restored order. 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(2))
        [2 means environment outside failset]
fail(message) = enveval(message,list(pop(failist)))
```

The function select defined below returns the first element of its argument set when first called; upon subsequent fails back to select, successive elements from set are returned. If set is exhausted, failure is propagated back. The code uses the fact that the binding environment saved by failset shares the variable fig with the instance of select which calls failset. The test of fig is reached in two ways: after a call on failset (in which *case fig* is false) *and* after a failure (in which case fig is true).

```
select(set,undolist) =
     progt  (fig)
s1:  if  null(set) then fail(undolist) [leave here and
                                        undo as specified]
     fig  —  false;
     failsetOT
     [fig is true iff we have failed to this point;  then
                                        set has been popped]
     if fig then go(sl);
     fig — true;
     returnTpop(set));
     end
```

Floyd,"[7] Hewitt,9 and Golomb and Baumert[8] have discussed uses for backtracking in problem solving. Sussman[18] has discussed a number of problems with backtracking. In general, it proves to be too simple a form of switching between environments. Use of the multiple process feature described above provides much more flexibility.

5. Coordinated Sequential Processes and Parallel Processing

It should be noted that in the model above, control must be explicitly transferred 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 (cooperating sequential) processes which synchronize only when required (e. g. , by means of Dijkstra's[4] 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 in a single processor system, 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 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). (A related technique is used by Premier,[14])

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

1) 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 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),

2) stop(pd) — halts the execution of the process specified by pd — PS removes the process from runnable queue. The value returned is an ed of the current environment of pd.

3) continue(pd) -- returns pd to the runnable queues.

4) status(pd) — value is an indication of status of pd.

5) 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 hands control back to PI (with enveval) having decremented the the semaphore count, or enters P1 on that semaphore's queue in PS's environment and switches control to a runnable process.

6) release(semaphore) -- this *Dijkstra V operator* increments the semaphore count; if the count goes positive, one process is moved from the semaphore queue (if any exist) onto the runnable queue and the count is decremented. It then hands control back to the calling process.

We emphasize that these six functions can be defined in terms of the control primitives of section 3.

Scheduling of runnable processes could be done by having each process by agreement ask for a time resource, i.e., obtain(time), 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 a timer interrupt mechanism. Interrupts are treated as forced

calls to environ (to obtain an ed for the current state), and then an enveval to the monitor process. The only problem which must be handled by the system in forcing the call to environ is making sure the interrupted process is in a clean state; that is, one in which basic communication assumptions about states of pointers, queues, buffers, etc. are true (e.g., no pointers in machine registers which should be traced during garbage collection). This can be ensured if asynchronous hardware interrupts perform only minimal necessary operations, and set a software interrupt flag. Software checks made before procedure calls, returns and backward jumps within program will ensure that a timely response in a clean state will occur.

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; however, 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 must be uninterruptible, e.g., at the highest priority level.

Obtaining a system of cooperating sequential processes as an extension of the primitives has a number of desirable attributes. Most important, perhaps, it allows the scheduler to be defined by the user. When parallel processes are used to realize a breadth-first search of an or-graph, there is a significant issue of how the competing processes are to be allotted time. Provision for a user supplied scheduler establishes a framework in which an intelligent allocation algorithm can be employed.

Once a multi-process supervisor is defined, a variety of additional control structures may be readily created. As an example, consider multiple parallel returns — the ability to return from a single activation of a module G several times with several (different) values. For G to return to its caller with value given by val and still continue to run, G simply calls process(val, 1,2). Then the current G and the new process proceed in parallel.

6. Conclusion

In providing linguistic facilities more complex than hierarchical control, a key problem is finding a model that clearly exhibits the relation between processes, access modules, and their environment. This paper has presented a model which 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 provide a small basis on which one can define almost all known regimes of control.

Although not stressed in this paper, there is an implementation for the model which is perfectly general, yet for several subcases (e.g., simple recursion and backtracking) this implementation is as efficient as existing special techniques. The main ideas of the implementation are as follows (cf. [2] for details). The basic frame and frame extension are treated as potentially discontiguous segments. When a frame extension is to be used for running, it is copied to an open stack end if not there already, so that ordinary nested calls can use simple stack discipline for storage management. Reference counts are combined with a count propagation technique to ensure that only those frames are kept which are still in use.

Thus, the model provides both a linguistic framework for expressing control regimes, and a practical basis for an implementation. It is being incorporated intoBBN-LISP.19

7. Acknowledgments

This work was supported in part by the Advanced Research Projects Agency under Contracts DAHC 15-71-00088 and F19628-68-0-0379, and by the U.S. Air Force Electronics Systems Division under Contract F19628-71-C-0173. Daniel Bobrow was at Bolt Beranek and Newman, Cambridge, Massachusetts, when many of the ideas in this paper were first developed.

References

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

[2l Bobrow, D.G. and Wegbreit, B. "A Model and Stack Implementation of Multiple Environments," BBN Report No. 2334, Cambridge, Mass., March 1972, to appear in CACM.

[3] Burge, W.H. "Some Examples of the Use of Function Producing Functions," Second Symposium on Symbolic and Algebraic Manipulation, AC:M, 1971.

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

[5] Dijkstra, E.W. "Recursive Programming," Numerische Mathematik 2 (I960), 312-318. Also in Programming Systems and Languages, S. Rosen (Ed.), McGraw-Hill, New York, 1967.

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

[7] Floyd, R.W. "Non-deterministic Algorithms," J. ACM, 14 (October 1967), pp. 638-644.

[8t Golomb, S.W. and Baumert, L.D. "Backtrack Programming," J. ACM, 12 (October 1965), pp. 516-524.

[9! Hewitt, C. "PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot," *in* Artificial Intelligence, Washington, D.C., May 1969.

[10] Johnston, J.B. "The Contour Model of Block Structured Processes," in Tou and Wegner, Proc. Symposium on Data Structures in Programming Languages. SIGPLAN Notices, Vol. 6, No. 2, pp. 55-82.

[11] McCarthy, J., et al. Lisp 1. 5 Programmer's Manual, TheM.I.T. Press, Cambridge, Massachusetts (1962).

[12] Moses, J. "The Function of FUNCTION in LISP," SIGSAM Bulletin, No. 15, (July 1970), pp. 13-27.

[13j Prenner, C,, Spitzen, J. and Wegbreit, B. "An Implementation of Backtracking for Programming Languages," submitted for publication, ACM-72.

[14J Prenner, C. "Multi-path Control Structures for Programming Languages," Ph.D. Thesis, Harvard University, May 1972.

[151 Quam, L. LISP 1. 6 Reference Manual, Stanford AI Laboratory.

[16] 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.

[17] Rulifson, J. et al. "QA4 - A Language for Writing Problem-Solving Programs," SRI Technical Note 48, November 1970.

[18] Sussman, G.J. "Why Conniving is Better than Planning," FJCC 1972, pp. 1171-1179.

[19] Teitelman, W., Bobrow, D., Murphy, D., and Hartley, A. BBN-LISP Manual. BBN, July 1971.

[20] Tou, J, andWegner, P. (Eds.), SIGFLAN Notices — Proc. Symposium on Data Structures in Programming languages. Vol. 6, No. 2 (February 1971)

[2Il van Wijngaarden, A. (Ed.). Report on the Algorithmic Language ALGOL 68, MR 101, Mathematisch Centrum, Amsterdam (February 1969).

[22] Wegbreit, B, "Studies in Extensible Programming Languages" Ph.D. Thesis, Harvard University, May 1970.

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

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

[25] Weizenbaum, J. "The Funarg Problem Explained," M.I.T., Cambridge, Mass., March 1968.

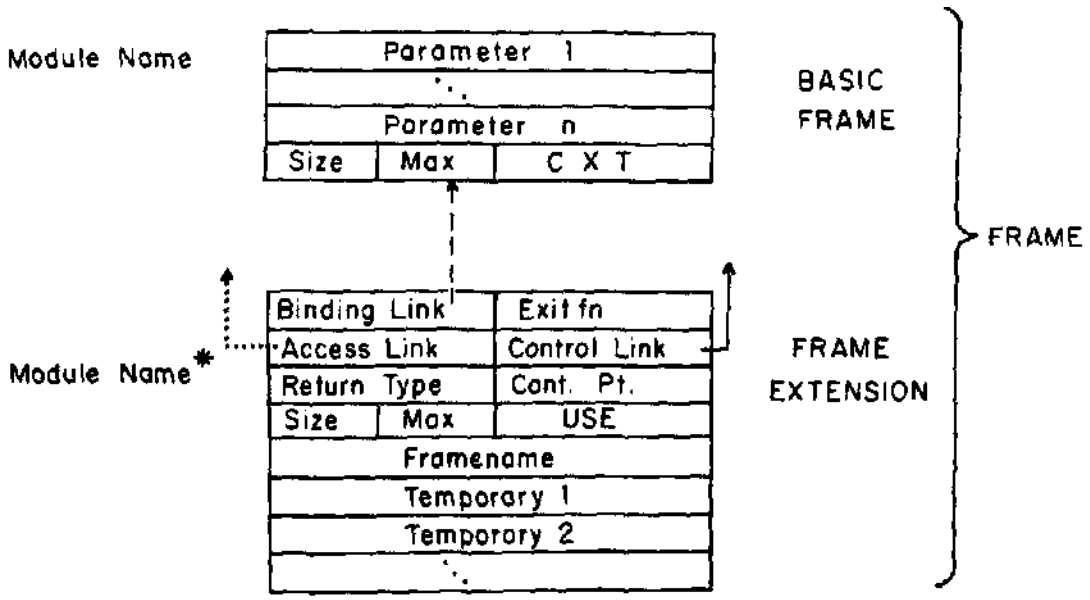

FIG. 1 GENERAL FRAME STRUCTURE

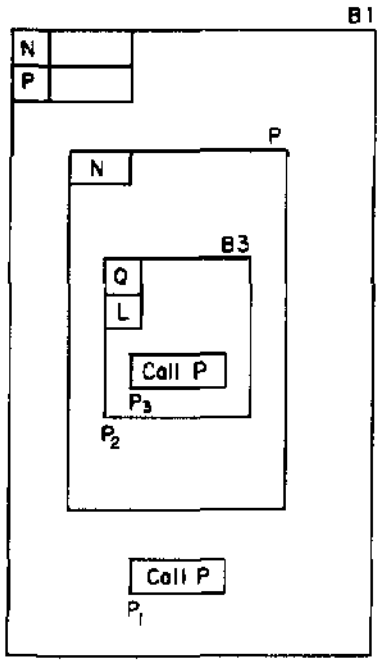

FIG. 2a (from Johnston)
BLOCK B1 WITH LOCALS N, P
PROCEDURE P WITH NEW VARIABLE N
BLOCK B3 WITH LOCALS Q, L
CALLS TO P WITHIN B1 AND B3

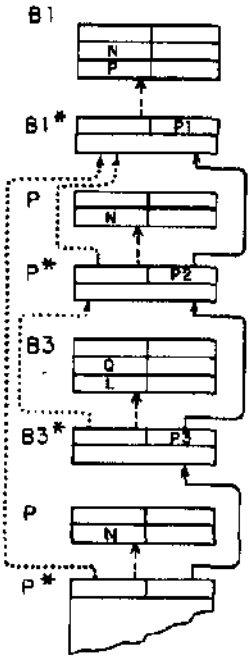

FIG. 2b SNAPSHOT OF FRAME STRUCTURE STARTING AT B1, CALL TO P, ENTER B3, CALL TO P

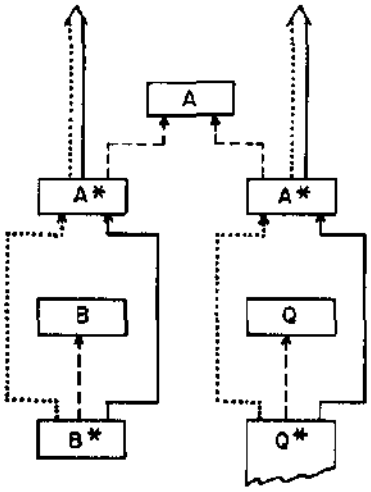

FIG. 3a COROUTINES SHARING ANCESTOR MODULE A, Q IS ACTIVE

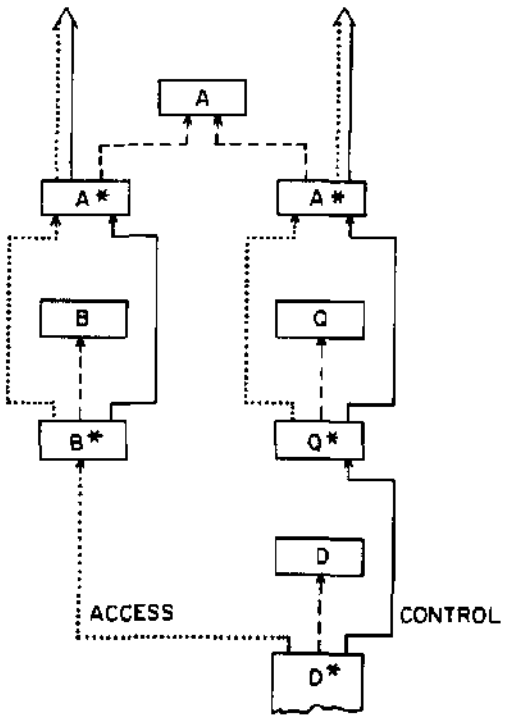

FIG. 3b COROUTINE Q EVALUATING FUNCTION D IN ACCESS CONTEXT OF B*

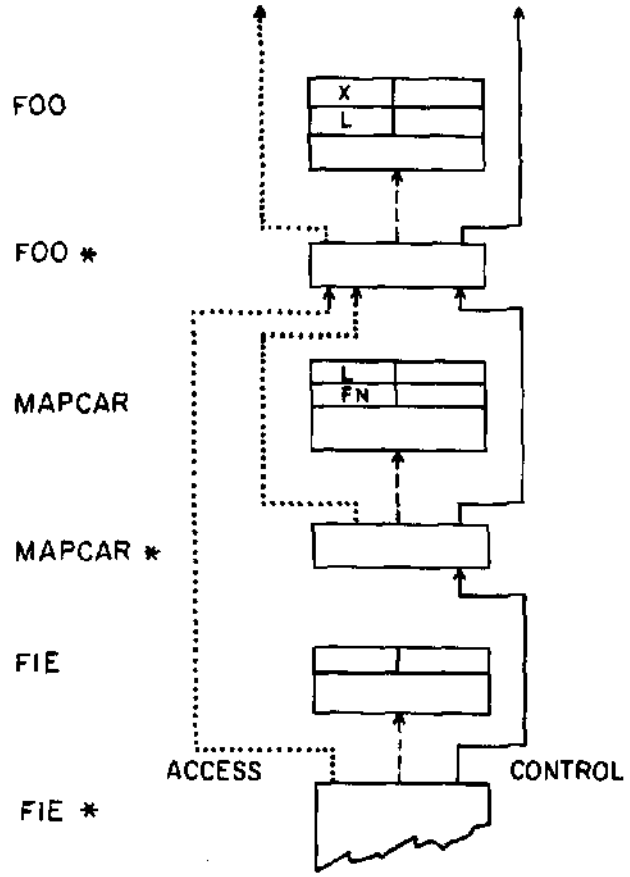

FIG. 4 APPLICATION OF A FUNCTIONAL ARGUMENT