BLOCK STRUCTURE: RETENTION OR DELETION?*

(EXTENDED ABSTRACT)

by

Daniel M. Berry
Center for Computer and Information Science
Division of Applied Mathematics
Brown University, Providence, R.I. 02912

Abstract: The question as to the correct block exit strategy, retention or deletion, is resolved by formally comparing the contour model and the stack model, each of which implements one of the strategies, to the copy rule, a formal definition of block structuring.

1. INTRODUCTION

Block structure was introduced with the programming language Algol 60 [Nau 60,63] primarily to provide the ability to define local variables. Since then the notion of block structure has been generalized to a full spectrum of block structured languages, including the well-known Algol 68 [vWn 69] and PL/1 [LW 69, Bee 70], and the author's Oregano [Bry 71].

This paper is concerned with a particular semantic problem regarding the execution of programs in nearly all block structured languages. Before stating the problem, we very briefly list the major syntactic and semantic features of a typical generalized block structured language.

1.1 Syntax

A program in a block structured language is generally a single block. A block has a declaration list and a statement list. A statement, which is optionally labelled, is either an assignment statement, a conditional statement, a procedure call, a goto statement or a (nested) block.

A procedure is similar to a block. It has a parameter specification list and a body which consists of a single statement (which of course may be a block). In its most general form an assignment statement has a left-hand side which may be a simple variable, a subscripted variable, or a pointer variable with indirection specified (to allow assignment to a cell pointed to by a pointer) and a right-hand side which may be a variable, some arithmetic, boolean or pointer expression, a procedure or a label. Thus we admit assignments of pointers, labels and procedures.

We observe that blocks and procedures may be arbitrarily nested within each other provided that the nesting is total; i.e., no overlapped blocks and/or procedures.

All identifiers used in a program must be declared in some block or be specified as a parameter of some procedure. We shall use the term "declaration" to mean both declarations and parameter specifications. A declaration of an identifier is said to have a scope which includes all statements, blocks and procedures nested inside the declaring block or procedure except for those blocks and procedures in which the identifier is redeclared. The use of an identifier is said to identify the declaration in whose scope the use lies; every legal use of an identifier must identify some declaration.

*This work was supported in part by NSF Grant GP 7347.

We have described a generalized block structured language which extracts the essential features from all actual block structured languages. With one exception, none of the actual languages listed above are as general in their assignment statement (they have various restrictions which happen to result from the semantic problem that we will discuss). Hence, in general discussions and in formal statements we use the term "block structured language" to denote block structuring in its fullest generality. However, in illustrations and motivating examples we use programs written in actual block structured languages.

1.2 Semantics

The execution of a program in a block structured language begins with entry of the outer block and proceeds sequentially through block entries, statements and block exits with the following exceptions:

1) When the statement is a goto, execution continues with the labelled statement. This may result in exit of blocks and procedures lying between the goto and the labelled statement.

2) When a procedure is called, the procedure is entered, parameters are passed, and execution continues at the beginning of the procedure body. If the end of the body is reached, execution continues at the statement following the call or at the call statement itself if the procedure returns a value.

1.3 The problem

The action upon entry to a block or a procedure is well understood:

A cell is allocated for each identifier which is declared in the entered block or procedure. In the case of a procedure entry the cells are initialized with actual parameter values (thus we are assuming call-by-value).

However, the action upon exit from a block or a procedure is not as well understood (this includes exit by way of a goto). There are two choices for defining the action:

1) All identifiers declared in the exited block or procedure become invalid for use in statements. Furthermore, all cells allocated for these invalid identifiers are automatically deallocated.
2) All identifiers declared in the exited block or procedure become invalid for use in statements. However, a cell is deallocated only when the cell is no longer accessible by any chain of

pointers.

The first choice, characterized by automatic deallocation, will be referred to as the deletion strategy. One example of an implementation of block structuring which follows the deletion strategy is E.W. Dijkstra's stack model, SM [Dij 60, BR 64]. In SM, cells for the identifiers declared in a block or procedure are pushed into the top of a run-time stack upon entry and are popped from the stack upon exit. The second choice, characterized by only possible deallocation and thus possible non-deallocation, will be referred to as the retention strategy. One example of an implementation of block structuring which uses the retention strategy is J.B. Johnston's contour model, CM [Joh 71]. In CM cells for the identifiers declared in a block or procedure are allocated from free storage upon entry and they are deallocated only when they become inaccessible by any chain of pointers.

This paper is an investigation of the consequences of the two strategies and it attempts to determine by formal methods which of the two strategies is correct. For the remainder of the paper the two strategies, retention and deletion, will be discussed in terms of the two models, CM and SM, which implement the strategies.

We first define the notion of an information structure model and give a more detailed, but somewhat informal* definition of the two models as special cases of information structure models. The description of the models includes specific program examples, some of whose executions differ in the two models. After conveying a little historical perspective, the copy rule is given as a formal definition of block structuring. We observe that the copy rule appears to have retention; this suspicion is verified by proving the contour model equivalent to the copy rule. Finally we give conditions for equivalence of the stack model to the copy rule.

2. INFORMATION STRUCTURE MODELS

All of the models for execution of block structured languages presented in this paper are special cases of information structure models [Weg 68,70a,b,MW 71,McG 70b,71].

Definition 1: An information structure model is a triple $M=(I,I_0,F)$ where I is a countable set of snapshots (information structures), $I_0 \subset I$ is the set of initial snapshots, and F is a finite set of transformations, $f:I \to I \cup \{\phi\}$, each of which transforms a given snapshot to another snapshot or to the null set.

Typically, snapshots will consist of such information structures as cells, stacks, lists, trees, etc. Each initial snapshot will contain some representation of some program which may be executed in the model. A transformation will reflect statement execution. Since we are modelling sequential program executions, in which no parallel tasks are created, the transformations are deterministic.

Definition 2: In an information structure model $M=(I,I_0,F)$, a transformation $f \in F$ is said to be applicable to a snapshot $S \in I$ if $f(S) \in I$ and not applicable to S if $f(S)=\phi$.

We describe the execution of a single program as a computation.

Definition 3: A computation in the information structure model $M=(I,I_0,F)$ is defined to be a sequence of snapshots $S_0,S_1,\ldots,S_{j-1},S_j,\ldots$ each an element of I, where $S_0 \in I_0$ is the initial snapshot and each snapshot S_j is obtained from the previous snapshot S_{j-1} by application of some transformation $f \in F$; i.e., $S_j=f(S_{j-1})$. If there exists a snapshot S_n such that no transformation of F is applicable, i.e. for all $f \in F$, $f(S_n)=\phi$, then the computation is said to halt at the final snapshot S_n.

Since transformations are deterministic, each computation will be a function of the initial snapshot.

Definition 4: The computation in $M=(I,I_0,F)$ arising from the initial snapshot S_0 is denoted by $M(S_0)$.

The information structure models given are for program execution. Thus we assume that for each model M there is a compile function compile_M which converts a program p into an initial snapshot.

Definition 5: For each information structure model $M=(I,I_0,F)$ for executing programs in a block structured language L, there exists a compiling function, $\text{compile}_M: L \to I_0$.

Thus the computation in M of the program p in L is denoted by $M(\text{compile}_M(p))$.

3. THE STACK MODEL AND THE CONTOUR MODEL

The stack model SM and the contour model CM may be defined as information structure models. The two models are quite similar and differ only in the structure of one component of a snapshot. Therefore we describe the two models in parallel.

3.1 Components of snapshots

In both SM and CM, each snapshot consists of three components, the algorithm, the record of execution (or simply record) and the processor, Π.

The algorithm is a program with the lines numbered for reference purposes.

The record of execution consists of all cells which are allocated during the record of execution. The two models distinguish themselves by the form that their records of execution take. In SM, the record of execution is a stack of activation records, each called an AR. Each AR consists of all cells allocated for one block or procedure entry. In CM, the record of execution is a set of contours allocated from free storage. Each contour consists of all cells allocated for one block or procedure entry.

In both models the processor Π models a

*Complete formal definitions of the model in the Vienna Definition Language [LW 69,HJ 70], besides being unreadable, would by themselves use up the allotted pages.

processing unit and consists of two pointers, an instruction pointer, ip, and an environment pointer, ep. The processor's ip points to the currently executed statement of the algorithm. The processor's ep points by a chain of pointers to a subset of the record of execution which constitutes the processor's accessing environment.

3.2 Transformations

Both SM and CM have the same single transformation f which may be described informally as:

1) fetch the statement pointed to by the processor's ip
2) sequence the processor's ip to point to the next statement.
3) execute the fetched statement using the cells of the processor's accessing environment.

Step 3 is broken up into sub-cases, each corresponding to a particular statement type.

In the following sections we illustrate the computation of four programs in SM and CM. These illustrate in more detail the components of a snapshot, initial snapshots, and the transformation sub-cases for particular statement types. Since the models are similar, the snapshots of a computation of a program are shown for the two models in parallel. (In fact, the snapshots will share the algorithm.) This focuses attention on the similarities and differences between SM and CM and will also give a feel for forthcoming proof techniques.

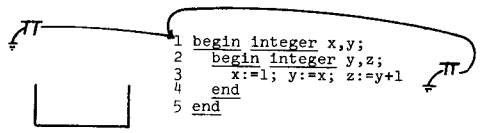
3.3 Block entry and exit

Consider the following simple Algol 60 program:

```
1   begin integer x,y;
2       begin integer y,z;
3           x:=1;y:=x;z:=y+1
4       end
5   end
```

There is an outer block which declares x and y to be integer variables and an inner block which declares y and z to be integer variables.

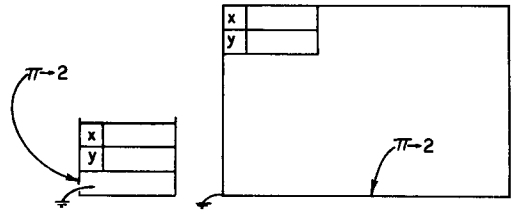
Let us consider the initial snapshot for both models. The algorithm is the program. For SM, the record consists of an empty stack. For CM, the record is empty. In both cases there is a processor whose ip points to the first statement in line 1 and a null ep (grounded arrow).



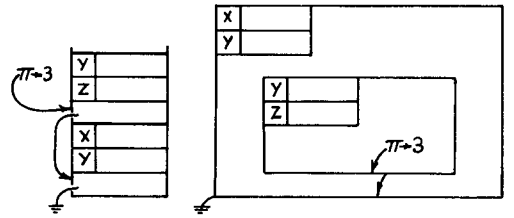
Record of SM | Algorithm of SM and CM | Record of CM
SNAPSHOTS 0 (INITIAL)

Since the algorithm is fixed we shall not show it any more. We also denote an ep pointing to line 1 as "→1".

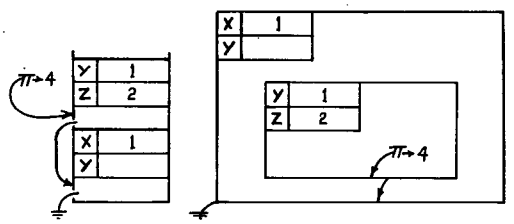
The statement in line 1 is fetched and the ip is sequenced to point to line 2. This sequencing takes place during every statement execution and will not be mentioned any more. The fetched statement, begin integer x,y, is a block entry. In SM, an AR is allocated on top of the stack. The AR consists of one cell for each identifier declared in the block and a cell for a static link. In CM a contour is allocated with a declaration array with a cell for each identifier declared in the block; it also has a static link. In both cases the static link is made to be a copy of the processor's current ep. Finally, the processor's ep is then set to point to this new AR (at the base) or to this new contour.



The statement in line 2 is begin integer y, z, a block entry. In SM an AR with cells for y and z is pushed on top of the stack. In CM a contour with cells for y and z is allocated. In each case the static link is made to be a copy of the processor's ep and then the processor's ep is set to point to the new AR as contour.



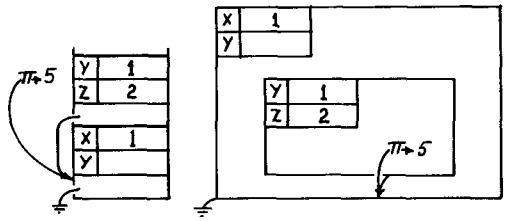
Line 3 has three assignments, x:=1; y:=z; and z:=y+1. In both models the cells for y and z may be found in the AR or contour pointed to by the processor's ep. To find the cell for x the processor follows the static link of that AR or contour to the other AR or contour; there it finds a cell for x. As a result x=1, y (of the top AR or inner contour) = 1 and z=2.



Now a processor's accessing environment may be precisely defined. In both models, the processor's accessing environment is defined to be the set of ARs or contours which may be reached from the processor's ep by a chain of static links. For any identifier the cell that is used is the first one reached by following this static chain from the processor's ep. In SM the cell used is the topmost cell for the identifier in the accessing environment. In CM we have been careful to topologically nest a given contour or the processor

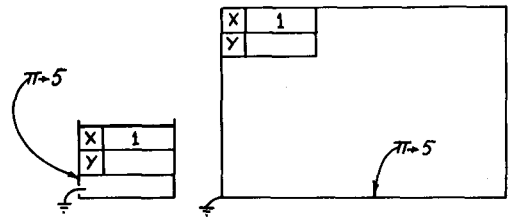
inside the contour to which its static link or ep points. Therefore the processor's accessing environment consists of all contours enclosing the processor. The cell used for any identifier is the innermost one for that identifier. Notice that by this environment search scheme the cells corresponding to the proper declarations have been used and, in particular, the cell for y in the bottom AR or outer contour corresponding to the outer block declaration has not and cannot be used.

The processor's ip now points to the end of the inner block. The processor's ep is made to be a copy of the static link of the AR or contour pointed to by the processor's current ep.



SM SNAPSHOTS 4 CM

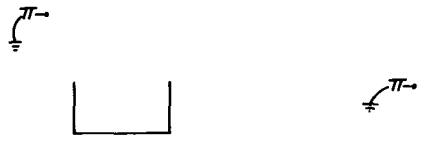
In SM, the changing of the ep has the effect of popping the stack. Since the AR for another block entry would go in the space occupied by the popped AR, that AR is effectively* deallocated. In CM the change in the ep has made the inner contour inaccessible to the processor. Since the inner contour can no longer affect the computation, it may be deallocated**. Note well the difference in criteria for deallocation in the two models. We show snapshots 4 again with the deallocated AR and contours erased.



SM SNAPSHOTS 4' CM

Line 5 is another block exit. The processor's ep is set to a copy of the null static link of the AR or contour pointed to by the current value of the ep. The processor's environment is now null. In SM the remaining AR is popped, leaving an empty stack. In CM the remaining contour is no longer accessible so it is deallocated, leaving a null record of execution (see snapshots 5).

In both models the processor has run out of instructions, leaving a null ip. Thus there is no transformation applicable. The computations in SM and CM both halt with snapshot 5 as the final snapshot.



SM SNAPSHOTS 5 CM

For the above example program, SM and CM seem to give rise to the same computation. Indeed, the only place the two models differed (besides in the obvious pictorial and vocabulary differences) is in the description of what happened after block exit. In this program, there was deallocation in both models.

The reason the computations did not differ with the two models is that we were careful to use an Algol 60 program which does not have pointer assignment and unrestricted label and procedure assignments*. In languages such as Oregano, PL/1 and Algol 68, which do have less restricted pointer, label and procedure assignments, the choice of exit strategy becomes critical. As we show in the three following sections, there exist programs which give rise to different computations in the two models.

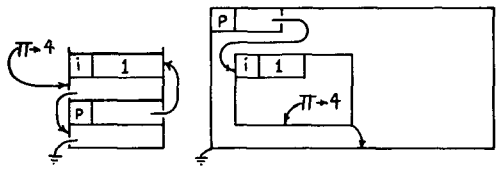
3.4 Pointer assignments

The first example is an Oregano program in which a tricky pointer assignment is made.

```
1   (ptr int p;
2      (int i←1;
3         p←@i
4      )
5      print(p*);
6      (real a←2.1;
7         print(p*)
8      )
9   )
```

In Oregano blocks are delimited by parentheses. In this program p is declared to be a variable of mode pointer to an integer. In line 3 @ means "address of" so that p is assigned a pointer pointing to the cell for i. In lines 5 and 7 * means indirection, so that the value pointed to by p is printed.

After entry to the outer and the first inner blocks and execution of p←@i, the snapshots in SM and CM are:



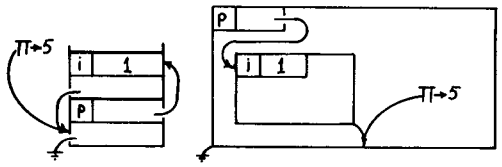
SM SNAPSHOTS 3 CM

In line 5 the inner block is exited. The processor's ep is set to point to the bottom AR or to the outer contour.

*We say "effectively" because the contents of the AR may not be erased until it is written over.

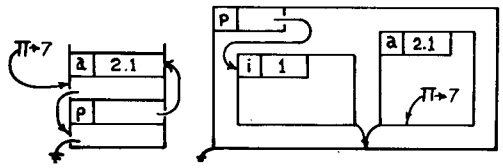
**A contour is one big cell which has as an integral part its declaration array. A contour is allocated and deallocated as a unit.

*In Algol 60 labels and procedures may be passed as parameters; this restricted form of assignment, as we shall see, causes no problems.



SM SNAPSHOTS 4 CM

In the SM case the top AR has been officially deallocated. Thus the indirection on p in line 5 should fail. However, for the skeptical reader we grant that the contents of the deallocated AR have not yet been erased, so that the indirection on p works and 1 is printed. In CM the inner contour with the cell for i is still accessible via the pointer in p which lies in the processor's environment. Indirection on p succeeds and 1 is printed. The two computations appear the same so far but we smash this supposition to smithereens with the block entry in line 6. In SM an AR with a cell for real a initialized to 2.1 is pushed into the stack in the space occupied by the deallocated AR. The pointer in p now points to the cell for a (assuming reals take as much space as integers). In CM, however, the new contour with the initialized cell for a is placed in some place other than where the retained inner contour is (remember, the contours are allocated from free storage). The pointer in p still points to the cell for i.



SM SNAPSHOTS 6 CM

Line 7 reads print(p*). In SM, garbage would be printed as the integer conversion routine is applied to the real value 2.1 pointed to by p. In CM, since p still points to the cell for i, the integer 1 is printed. The computations in SM and CM differ for this program.

3.5 Label assignments and gotos

The following PL/1 program shows how assignment of labels can make the choice of exit strategy critical and make the computations in SM and CM differ.

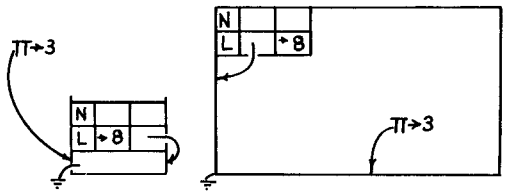
```
1   MAIN: PROC OPTIONS(MAIN);
2       DECLARE N LABEL;
3         BEGIN;
4            N=M;
5            GOTO L;
6            M: RETURN;
7         END;
8        L:GOTO N;
9       END;
```

A label constant identifier is assumed to be declared in the innermost block in which the identifier appears as a statement label. For example, the label constants L in line 8 and M in line 6 are assumed to be declared in the main procedure (block) and in the inner block respectively. Identifiers are declared to be label variables in an explicit declaration, as is the identifier N in line 2.

To be able to assign labels, labels must have values. A goto may be thought of as specifying a new "site" of execution for the processor. Since both an ip and an ep are required to completely specify a processor, a label value must also consist of an ip and an ep.

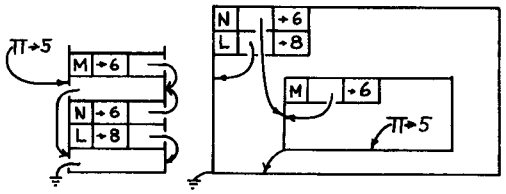
Just as a label constant is implicitly declared, it is also implicitly initialized upon entry to the block in which it is assumed to be declared. Its ip points to the labelled statement in the algorithm and its ep points to the contour allocated for the block.

Let us now consider the execution of the PL/1 program in SM and CM. As the main procedure is entered an AR and a contour are allocated with cells for explicitly declared N and implicitly declared L. In both models the cell for L is initialized with an ip pointing to line 8 and an ep pointing to the newly allocated AR or contour. Note that the ep of L is a copy of the processor's ep after entry. (To avoid clutter of the snapshots, in SM the ep is on the right-hand side of a cell but in CM the ep is on the left-hand side. There should be no confusion since the ip has a number in it.)



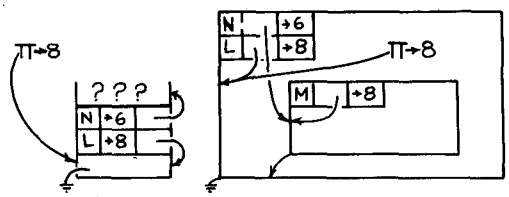
SM SNAPSHOTS 1 CM

Similarly, upon entry to the inner block an AR and a contour are allocated with an initialized cell for the label constant M. Then, in line 4, N=M results in copying both the ep and ip of M into the cell for N in the bottom AR or the outer contour.



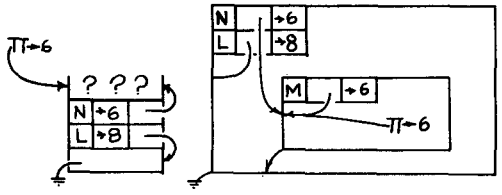
SM SNAPSHOTS 3 CM

GOTO L in line 5 merely results in copying the ep and ip of the label L into the ep and ip of the processor. Now the ep of the processor points to the bottom AR or to the outer contour. In SM the top AR has been effectively popped. It would be easy enough to enter another block to erase the current contents of the AR, so we indicate uncertainty as to the contents of the space vacated by the AR. In CM the inner contour is pointed to by the ep of the label N which is in the processor's environment. Hence the inner contour is retained.



SM SNAPSHOTS 4 CM

The processor's current instruction is GOTO N. As before, the ep and ip of N are copied into the ep and ip of the processor. In the CM case the ep points to a popped AR. As we have seen, it is easy to enter another block and force the wrong AR to become part of the label's environment. Therefore we say that the GOTO fails. In CM, on the other hand, the ep now points to the properly retained contour and the processor has regained the proper environment. The GOTO succeeds.



SM SNAPSHOTS 5 CM

The computations in SM and CM arising from this program could clearly be made to differ. They would differ because of the deletion strategy of SM on the one hand and the retention strategy of CM on the other hand.

3.6 Procedure assignments and calls

The following Algol 68 program has a procedure assignment which causes SM and CM to execute the program differently.

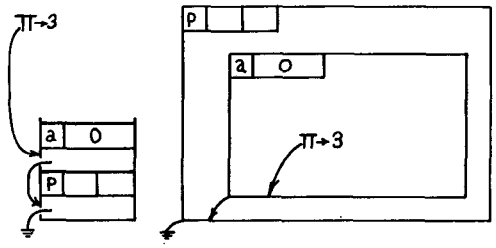
```
1    begin proc(int)p;
2          begin int a:=0;
3                p:=((int i):
4                   a:=i+a);
5                p(3);
6          end
7          p(2)
8    end
```

In this program, p is declared to be a procedure variable to which a procedure, which accepts an integer parameter and returns no result, may be assigned. In line 3, p is assigned such a procedure; its parameter is i and its body is the assignment a:=i+a.

Scope rules extend to procedure bodies so that in this case the non-local a in the body identifies the a declared in the inner block.

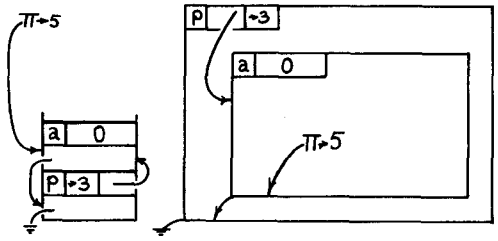
Procedure values must therefore consist of an ip to point to the code and an ep to point to the enviornment in which non-locals may be accessed at call time.

Let us follow the snapshots arising from execution of this program in SM and CM. Entry to the outer block results in allocation of an AR or contour with a cell for p. Entry of the inner block results in allocation of an AR or contour with a cell for a initialized to 0.



SM SNAPSHOTS 2 CM

Line 3 is the assignment of the procedure body to the variable p. The cell for p gets an ep which is a copy of the processor's ep and an ip pointing to the entry point of the procedure.



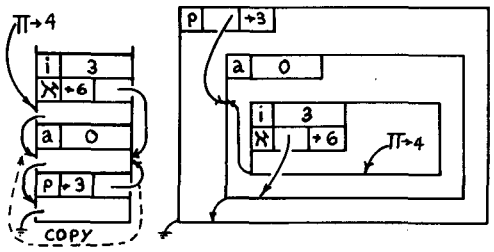
SM SNAPSHOTS 3 CM

Line 5 is a call of p(3). An AR or contour with a cell for the parameter i initialized to 3 and a cell for a return label, ℵ*. The cell for ℵ is filled with the current ip and ep or the processor. Remember that the ip has already been sequenced to the next statement so that the return label's ip points to the statement immediately following the call. The ep points to the calling environment.

(In SM the ep points one AR down and is commonly referred to as the dynamic link.) The static link of the new AR or contour is set to be a copy of the ep of the called procedure. Then the processor resets its ep to point to the new AR or contour and resets its ip to point to the statement after the entry point, which is pointed to by the procedure's ip (see snapshots 4).

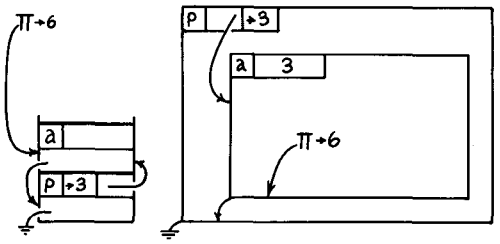
Because of the method of determining the static link of the AR or contour for the activation of the procedure, the non-locals whose scope includes the procedure body are now in the processor's accessing environment. Hence, after executing a:=i+a the processor has stored 3 in the cell for a in the middle AR or contour. Then at the end of the body, the return label ℵ is used in a goto to effect a return to the calling site at line 6. The processor's ep is reset to point to

*Since very few keypunches have the letter aleph, ℵ, this is a safe identifier to use. It cannot possibly conflict with the scope of any programmer-defined identifier. The Algol 68 Report [vWn 69] uses ℵ for the same purpose.



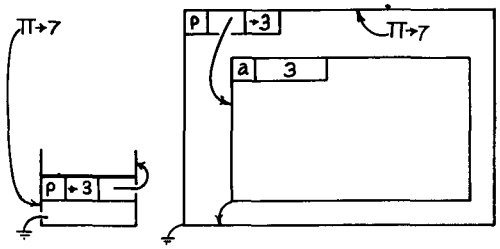
SM SNAPSHOTS 4 CM

the middle AR or contour. As a result, in SM the top AR is popped, and in CM the inner contour is deallocated because it is no longer accessible.



SM SNAPSHOTS 6 CM

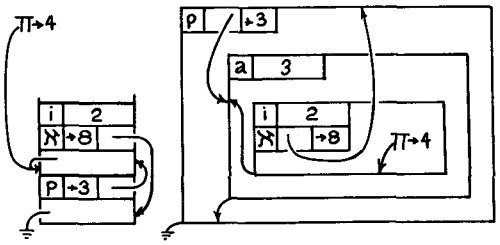
Now in line 6 the inner block is exited. In CM this pops the top AR (again, we say that this AR has been deallocated because one can always write over this AR with a new block entry). In CM the processor's ep points to the outer contour, but since p's ep points to the inner contour the inner is retained.



SM SNAPSHOTS 7 CM

Line 7 has a call of p(2). An AR or contour is allocated with cells for i initialized to 2 and a return label ↯. The static link of the AR or contour is made a copy of the ep of p. Finally, the processor sets its ep to point to the new AR and sets its ip to point to the first statement of the body (see snapshots 8).

In SM we have a very weird situation! The static link of the top AR points to itself. The processor's environment consists of only i and ↯ and a static chain search for any other identifier would loop forever. Missing from the processor's environment are the non-locals, a and p. In CM,



SM SNAPSHOTS 8 CM

because the innermost contour was retained, the processor's accessing environment is exactly as it should be. In particular, the environment is finite and includes i, ↯, a and p. Therefore as the processor attempts to execute a:=i+a, the two computations would differ; SM would loop infinitely and CM would continue on until termination.

The difference between SM and CM is clear now. SM follows the deletion strategy by virtue of the strict last-in-first-out order of deallocation imposed by the stack which is the record of execution. An AR must be deallocated upon exit to make room for any subsequent block entries. CM, however, is able to follow the retention strategy because it allocates contours from free storage. There is no compulsion to deallocate contours to make room for others because new contours may be allocated from wherever there is unused space. Thus it is the structure of the record of execution which makes the models differ and forces one to adopt the deletion strategy while permitting the other to adopt the retention strategy.

4. HISTORY

We now ask the inevitable question. Which is the correct exit strategy, retention or deletion?

Certainly, after seeing the above examples, we can say that retention is more general and more aesthetic than deletion. We were able to do more pointer, label and procedure assignment without diastrous and unexpected results in CM, which has retention, than in SM, which has deletion.

Historically the trend is towards the deletion strategy. The stack model was introduced in 1960 by E.W. Dijkstra [Dij 60] as a suggested implementation of Algol 60 (which was, of course, also introduced in 1960). Since then the stack model and deletion-oriented variants of it (the author knows of a free storage implementation that uses deletion [Brg 70]) have been almost universally adopted as the basis for block structured language implementation and design.

It appears that Algol 60 was very carefully designed to be implemented on a stack. As a result Algol 60 completely avoids pointers and restricts labels and procedures to be either constants which are initialized at declaration time or to be passed as parameters of procedures. We show later that under these restrictions retention and deletion strategies produce precisely the same

result in all computations. Thus, at the time the stack was introduced for implementing block structured languages it was not considered important to look for another model. Probably the distinction between the two strategies and their relationships to particular implementation models did not even occur to the designers of Algol 60. Indeed, even if the distinction did occur it was not necessary at that time to resolve the issue as to which strategy is correct.

However, in later block structured languages in which less restricted pointer, label and procedure assignments were available, a decision on the exit strategies had to be made. Algol 68 and PL/1 chose the deletion strategy and are both usually implemented on some variant of the stack model. Consequently both of these languages have restrictions against completely general pointer, label and procedure assignments. Algol 68 has provided for run-time checks to prevent these assignments. PL/1 warns the programmer that certain of these assignments will cause trouble but leaves the prevention of these assignments to the programmer.

On the other side of the coin, Oregano chose the retention strategy and was designed with the contour model as its underlying model. Consequently, Oregano has no restrictions on pointer, label and procedure assignments (other than those to ensure type matching).

With this historical perspective, we look to formal definitions of block structuring for insight as to which exit strategy is correct. The notions of blocks and scope of identifiers were borrowed from the lambda calculus [Chu 41]* to provide the Algol 60 programmer with a powerful programming tool, that of being able to write blocks and procedures without regard to identifiers used elsewhere, i.e. modularity. The first attempt at a formal definition of block structuring appeared in the Algol 60 report [Nau 60]. Certain essential paragraphs dealing with the treatment of identifiers were very poorly and incorrectly worded; c.f. 4.1.3, 4.7.3.1, 4.7.3.2, 4.7.3.3. Later, in 1963, the revised report [Nau 63] fixed up errors, some of which were based in inadequate regard for naming conflicts in the lambda calculus. Unfortunately, even in the revised form these paragraphs were poorly worded. It is no wonder that the Dijkstra stack model, which came out at the same time as the original report and which explained the report, was chosen as the standard for block structured languages. The unfortunate consequence of this has been the almost universal opinion that the deletion strategy is intrinsic to block structure.

Later several better statements of block structuring appeared in Lucas [Luc 70], Wegner [Weg 70c] and Henhapl and Jones [HJ 70]. The rules governing the treatment of identifiers upon block entry are collectively referred to as the copy rule.

*It is most revealing to observe that the pure lambda calculus allows a λ-expression (procedure) to return as its value another λ-expression (procedure) with a free variable (non-local) of the λ-expression which is bound to (identifies) the λ-variable (parameter)of the first; e.g., this λ-expression

$$\lambda x.\lambda y.x+y$$

The reader should convince himself that retention is necessary to retain the cell for x after λy.x+y is returned. McGowan [McG 70a,71] has shown that Landin's SECD machine [Lan 64] fails to compute all call-by-value λ-expressions because it behaves as if it followed the deletion strategy even though it has the necessary information to implement retention. Furthermore, McGowan has designed and proved correct a modified SECD machine which uses retention. Rubin [Rub 71] has devised and proved correct a contour model interpreter for the λ-calculus.

5. COPY RULE

In this paper we describe an information structure model for the copy rule, CR, which is a cross between the models used in [Weg 70c] and [HJ 70]*.

5.1 Components

In CR each snapshot consists of four components:

1) a stack of modified texts of the program being executed. The bottom of this stack contains an unmodified text of the program.
2) an ip which points directly to statements in the unmodified text in the bottom of the stack and which points indirectly to corresponding statements in modified texts higher up the stack.
3) a block entry count generator, BECG, which is incremented by 1 at each block or procedure entry. At block entry the current value of the BECG is used to subscript identifiers declared in the block to form unique names (addresses). A unique name has the property that it has never been used before in the computation.
4) a countably infinite two-dimensional storage component whose locations are addressed by ordered pairs of the form (id,n) where id is an identifier and n is an integer. The unique name id_n addresses location (id,n) of the storage.

5.2 Transformation

The single transformation of CR is described briefly:

1) fetch the statement in the topmost text of the program in the stack which is indirectly pointed to by the ip.

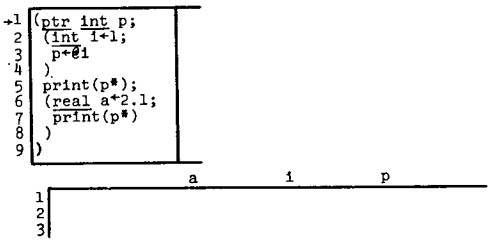
*These authors both refer to the problems arising from deletion combined with pointers, labels and procedures (cf. pp. 4-5 of [HJ 70] and p. 66 of [Weg 70 c]. However, these definitions were given to handle Algol 60. It was discovered by this author that Wegner's model has retention for pointers and procedures (he doesn't discuss labels). The model in [HJ 70] has retention for pointers but a simplification was made to the model for procedures and labels which was allowed by the restrictions of Algol 60 and which destroyed retention for procedures and labels. Also the machine has call by name.

2) sequence the ip to point to the next statement.
3) execute the fetched statement using the storage location addressed by the unique names appearing in the statement. (If a non-unique name is found there is an error, but this happens only if there is an undeclared identifier and thus will not occur for syntactically correct programs.)

5.3 Block entry and exit and pointers

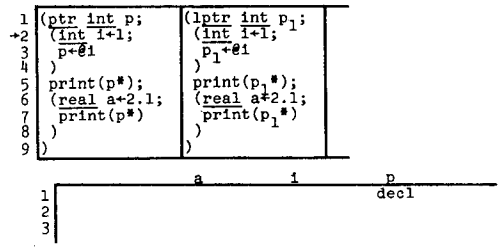
We informally describe the initial snapshot and statement execution phase of the transformation by showing the sequence of snapshots arising from execution of the Oregano pointer example of section 3.4.

The initial snapshot consists of a text of the program on the stack, the ip pointing to the first statement, the BECG set to 1, and the uninitialized storage component. (The stack is shown with the top to the right.)



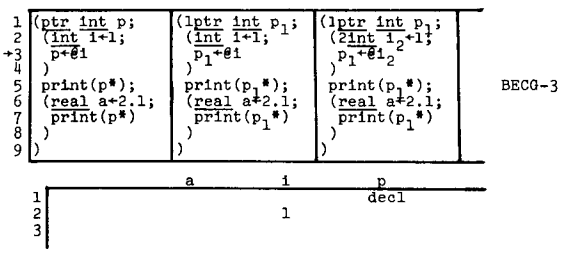
SNAPSHOT 0

The first statement is the entry of the outer block in which p is declared. A new text of the program is pushed into the stack. The new text is obtained by first copying the previous top-of-stack text; in that copy the current value of the BECG is used to subscript all instances in the entered block of each identifier (labels too, if any) declared in the block. Also the current value of the BECG is written immediately following the begin delimiter of the block (this is done merely to help the proof and in the actual computation this number is never used). Thus in this case each instance of p in the outer block is subscripted with 1 and after the open parenthesis is written 1. Then BECG is incremented by 1. Finally the storage location for each newly created unique name is marked "declared".



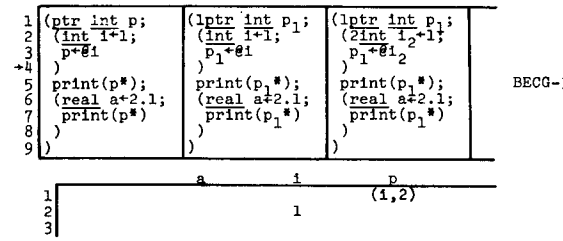
SNAPSHOT 1

The next statement is another block entry, int i←1. A new text of the program is pushed into the stack. In this new copy of text, each instance of i in the first inner block is subscripted by the current value of the BECG, 2, and the open parenthesis is followed by 2. The BECG is then incremented to 3. The declaration now reads int i_2←1. Since it is an initializing declaration it is not necessary to mark i_2 as declared. Instead 1 is stored into the location addressed by (i,2).



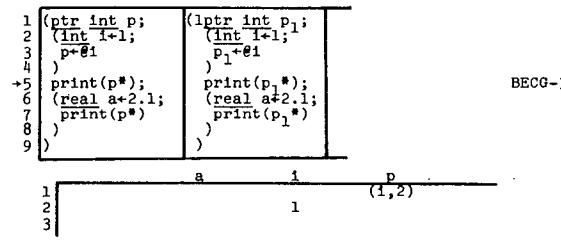
SNAPSHOT 2

Execution of the statement p_1←@i_2 results in storing the address of i_2, that is (i,2) into the location (p,1).



SNAPSHOT 3

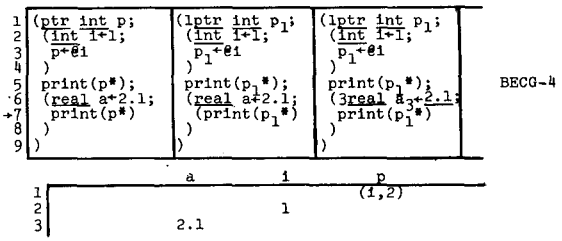
The block exit in line 4 results merely in popping the top text from the stack. Nothing else is changed except, of course, the ip.



SNAPSHOT 4

The ip points to print(p_1*). In the location (p,1) is stored (i,2). In the location (i,2) is stored 1. So 1 is printed. Note that even though the copy with i_2 has been popped, the indirection works because storage was untouched by block exit (ah ha! sayeth the reader!). Next the second inner block is entered. In this block a is declared and initalized to 2.1. A new text is pushed onto the stack; in this copy of the text each instance of a in the block is subscripted by 3, and the begin delimiter is numbered 3. The BECG is incremented to 4. Finally 2.1 is stored into (a,3) (see snapshot 6 below).

The statement print(p_1*) still works since nothing has happened to the contents of (p,1) and (i,2). This is because unique names are generated with the help of an ever-increasing BECG so that storage is never re-used for new declarations. The computation continues with two successive block exits and thus two successive pops of the



SNAPSHOT 6

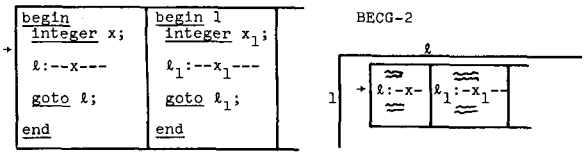
stack. The computation halts with the unmodified copy at the top of the stack, a null ip (run off the end of the program), the BECG still at 4 and the storage component still as above. So far it appears that CR follows the retention strategy.

5.4 Labels and procedures

It remains to complete the definition of CR by describing its treatment of labels and procedures.

Label values: a label value consists of
1) an ip pointing to the labelled statement
2) a copy of the entire stack just after entry of the block in which the label is assumed to be declared.

For example, the following snapshot was taken just after entry to an outer block in which the label constant ℓ is declared.



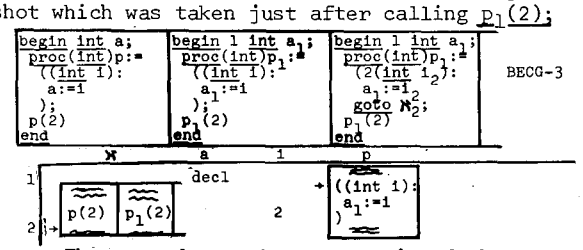
Goto statement: Replace the current ip and stack with that stored in the address of the label identifier. Continue execution.

Because the entire stack has been stored with the label value, it is always possible to recover the label's entire environment even if the current stack has a completely different set of modified programs (hence there is retention!).

Procedure value: a procedure value consists of:
1) an ip pointing to the entry point of the procedure text
2) a copy of the top text on the stack at the time the procedure is assigned.

Procedure call: the text of the program which is the procedure value is obtained and
a) the formal parameter identifiers in the procedure body (pointed to by ip of procedure value) are subscripted by the current value of the BECG. The begin delimiter is also numbered with the current value of BECG. The values of the actual parameters are stored in the appropriate formal parameter addresses.
b) the end delimiter of the procedure body is replaced by goto $\aleph_i$ where i is the current value of the BECG. A label value indicating the return site is stored into ($\aleph$,i). This new copy of the text is pushed onto the stack, the BECG is incremented by 1 and the ip is set to be a copy of that of the procedure value. Execution continues in the body.

As an example, consider the following snapshot which was taken just after calling $p_1(2)$;



The procedure value was assigned when the stack height was two. As a result the value stored in (p,1) has a copy of the second stack level. In this copy all non-locals inside the body have already been subscripted. Thus no matter when the procedure is called the proper instance of the non-local may be used (hence we have retention!). Note also that in (i,2) is stored the value of the actual parameter and in ($\aleph$,2) is stored a copy of the stack at call time and an ip pointing to the instruction after the call. The procedure return is accomplished merely as a simple goto.

We strongly suspect that the copy rule follows the retention strategy rather than the deletion strategy. We demonstrate this by proving CR to be equivalent to CM.

6. EQUIVALENCE OF INFORMATION STRUCTURE MODELS

First we define what it means for two information structure models to be equivalent for a block structure language.

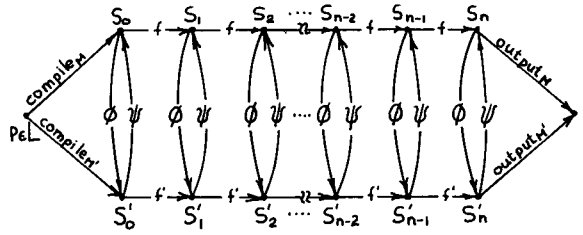
Definition 5. Two information structure models $M=(I,I_0,F)$ and $M'=(I',I_0',F')$ are said to be equivalent for a block structured language L if and only if for all programs $p \in L$
1) $M(S_0)$ halts if and only if $M'(S_0')$ halts, where $S_0=compile_M(p)$ and $S_0'=compile_{M'}(p)$.
2) If $M(S_0)$ halts then $output_M(final(M(S_0)))=output_{M'}(final(M'(S_0')))$, where final selects the final snapshot of a computation and $output_M$: $I \to 0 \subset I$, $output_{M'}: I' \to 0' \subset I'$ select from snapshots the output component or that which is being used as the equivalence criterion. (The output may indeed be the entire snapshot.)

The proof that two models are equivalent is usually very difficult to do in the literal method suggested by the definition. It is far easier to show by induction over the steps of the computations that the equivalence holds during the computations and thus holds at the ends of the computations if indeed they both halt. In this paper we use two such inductive proof techniques to prove a lemma and to prove the main theorem. The first technique is Lucas's twin machine method [Luc 68]. Essentially, the method involves defining one model called a twin machine which does the two computations in parallel in much the same manner as was done in the parallel descriptions of SM and CM. This works best when it is perfectly clear that the two computations will always be executing the same

instruction. It helps greatly if the two "parallel" computations of the twin machine actually share components of the snapshot such as the algorithm and the ip. If this is so it is immediate that the two "parallel" computations do execute the same instruction. Given this it is obvious that one computation halts if and only if the other halts. It remains to show that the output components of the two "parallel" computations remain the same in each snapshot.

When this sharing is not possible because of format difference, then the prime advantage of using the twin machine vanishes. Then it must be explicitly proved that the two computations execute the corresponding instructions. Instead we use a more general equivalence proof technique which is an adaptation of the McGowan mapping technique [McG 71, MW 71]. The adaptation used in this paper plays on the to-be-proved suspicion that the two computations execute corresponding instructions at the same rate. Schematically, we have computations in M and M' arising from execution of a program p.

computation $M(S_0)$ in $M=(I,I_0,F)$



computation $M'(S'_0)$ in $M'=(I',I'_0,F')$

To show models $M=(I,I_0,F)$ and $M'=(I',I'_0,F')$ equivalent for a block structured language L it suffices* to produce mappings

ϕ: I→I'

ψ: I'→I

such that for all programs pεL the following holds:

Let

$S_0=compile_M(p)$ and $S'_0=compile_{M'}(p)$

$M(S_0)=S_0,\ldots,S_i$, and

$M'(S'_0)=S'_0,\ldots,S'_i,\ldots$

Then

1a) $\phi(S_0)=S'_0$
$\psi(S'_0)=S_0$

2a) if $\phi(S_i)=S'_i$ and $S_i\neq final(M(S_0))$ then $\phi(S_{i+1})=S'_{i+1}$

2b) if $\psi(S'_i)=S_i$ and $S'_i\neq final(M'(S'_0))$ then $\psi(S'_{i+1})=S_{i+1}$.

3a) if $S_n = final(M(S_0))$ then $\phi(S_n) = final(M'(S'_0))$

3b) if $S'_n = final(M'(S'_0))$ then $\psi(S'_n) = final(M(S_0))$.

4a) $output_M(S_i) = output_{M'}\phi(S_i)$

4b) $output_{M'}(S'_i) = output_M\psi(S'_i)$.

In other words, we must find mappings from snapshots in each model to snapshots of the other. The first two conditions say that the initial snapshots must map to each other and that the succeeding snapshots of each computation map to each other. Given this, it is clear that each model will be executing its representation of the same statement at the same time and that representations of the same value are being stored in corresponding locations. The third condition says that the halting snapshot of each computation maps to each other. With this one can show that one computation halts if and only if the other does. The fourth condition says that output components of corresponding components are equal. With this it is immediate that the outputs of final snapshots are indeed equal. Thus the effective existence of two mappings from snapshots of each model to the other satisfying these conditions suffices to prove the models equivalent.

It may be noted that only the last two conditions are needed to show equivalence. However, the proof that the last two conditions hold for a particular set of mappings is done inductively. Typically (and in this paper) the first two conditions must be established in the inductive step for use in the proof that the last two are satisfied.

*The proof can be done merely by showing 1a,2a,3a, 4a and if $M'(S'_0)$ halts and $S'_n=final(M'(S'_0))$ then there exist S_i in $M(S_0)$ such that $\phi(S_i)=S'_n$. However, this requires finding ψ anyway so we use ψ from the beginning [McG 71].

7. EQUIVALENCE OF CM TO CR

To make it easier to identify contours in proofs we make a slight change in CM to produce a new model CM". We add to the processor a block entry count generator, BECG. In the initial snapshot the BECG reads 1. At each block or procedure entry the current value of the BECG is copied into the upper right-hand corner of the created contour and then the contents of the BECG is incremented by 1. We call this number in the upper right-hand corner the contour number. Intuitively the contour number may be thought of as an address, so that an ep could be a contour number. Since the contour number is never used in the computation this change cannot affect any computations. Therefore we state, without proof, this lemma:

Lemma 1: CM is equivalent to CM" for a block structured language L.

We define a third model CM' by adding to the definition of CM" the stipulation that no contours are deallocated. Since the only cells which may affect the computation are those that are accessible, leaving these inaccessible contours in the snapshots should not affect the computation. We prove this

Lemma 2: CM' is equivalent to CM" for a block structured language L.

Proof: Define a twin machine TM=($\mathfrak{A}$,$\mathfrak{A}_0$,$\mathfrak{F}$) whose snapshots $\mathfrak{f}\varepsilon\mathfrak{A}$ consist of

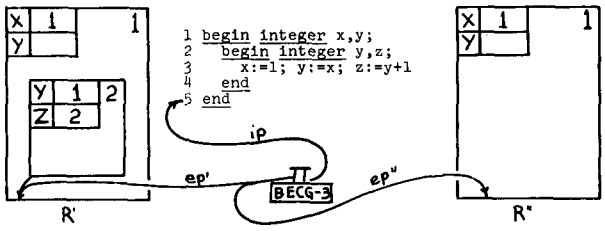
1) an algorithm
2) two records of execution: R' in the format of that of CM' and R" in the format of that of CM"
3) a processor π with one ip and two ep's, ep' pointing to the environment in R' and ep" pointing to the environment in R".

The initial snapshots $\mathfrak{f}_0\varepsilon\mathfrak{A}_0$ consist of an algorithm, a processor with its ip pointing to the first statement, a null ep', and a null ep".

The transformation $\mathfrak{f}\varepsilon\mathfrak{F}$ does the following:

1) fetch the statement pointed to by the processor's ip
2) sequence the ip to point to the next statement
3) execute the fetched statement in R" using ep" to locate the environment and deallocating contours as they become inaccessible; in R' using ep' to locate the environment and not deallocating any contours.

The following is a snapshot taken during TM's execution of the first example program of section 3.3 just after exit from the inner block.



SNAPSHOT 4

Now define a function find on {id|id is an identifier}X{ep|epε{ep',ep"}} such that

$$\text{find}(id,ep) = \begin{cases} n, \text{ if outward search of environment pointed to by ep finds the cell for } id \text{ in contour } n \\ \phi, \text{ if } id \text{ is not in environment pointed to by ep.} \end{cases}$$

Now it suffices to show by induction on the states of the computation that for all snapshots δ_i, for all id,

(1) find(id,ep') = find(id,ep")

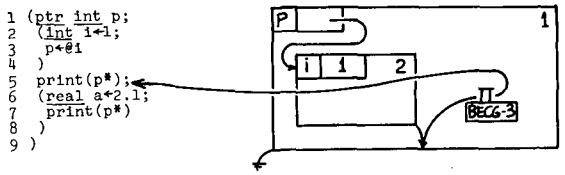
This says that an identifier is in one environment if and only if it is in the other and furthermore, if it is, it is found in like-numbered contours. Clearly (1) is true for δ_0. Given (1) is true for δ_i it is shown that it is true for δ_{i+1} by exhaustively showing that the change made by each statement preserves the truth of (1). For example, a block entry introduces the same set of identifiers to both environments. For any identifier id in this set find(id,ep')=find(id,ep")=number of the new contour allocated. The proof is tedious but trivial because the processor can only "see" cells which are accessible. Since the only difference between the two records is that R' has inaccessible contours still lying around and R" does not, the processor cannot detect any difference in the records of execution.

Corollary 1: CM is equivalent to CM' for a block structured language L.

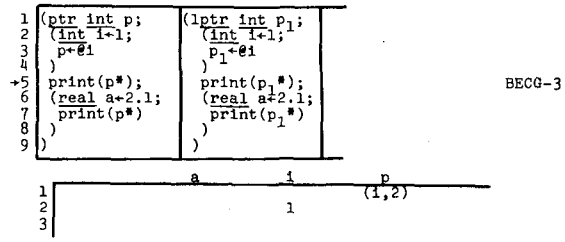
Now we show that CM' is equivalent to CR. To get an intuitive feeling for the proof, let us consider snapshots S_4' and S_4 from the execution of the pointer example of section 3.4 in CM' and CR. The snapshots (see below) are taken just after exit of the first inner block.

We can make a few observations about these snapshots which will help us in constructing mappings ϕ and ψ:

1) the BECG of S_4 is equal to the BECG of the processor of S_4'.



S_4' in CM'(S_0')



S_4 in CR(S_0)

2) the stack bottom text of S_4 is the same as the algorithm of S_4'.
3) the ip of S_4 points to the "same" instruction as does the ip of the processor in S_4'.
4) the ip of S_4 points to a particular statement s in the modified text at the top of the stack. Statement s is nested immediately within some modified block or procedure b. At the beginning of the text of b is a number m which was obtained from the BECG at the time b was entered. In S_4' the ep of the processor points to a contour numbered m.
5) the number of columns of the storage of S_4 which have been used is equal to the number of contours in S_4' which is equal to the BECG of either minus 1.
6) the contours which are part of the current environment in S_4' are those which are numbered the same as those blocks of the stack top text in S_4 which have been modified.

Theorem 1. CM' is equivalent to CR for a block structured language L.

Proof: Since the two models do not execute programs in the same format we use the McGowan mapping technique. We exhibit mappings ϕ, which maps a snapshot S of CR to the corresponding snapshot S' of CM', and ψ, which maps a snapshot S' of CM' to the corresponding snapshot S of CR. First we informally describe the mapping $\phi(S)=S'$.

A. We form the contours of S' from the storage component of S. From the bottom-of-stack text in S we form the finite set ID={id|id is used in the program}$\cup\{\aleph\}$. Now for each i, $1\leq i<$BECG-1, do 1-4:

1) create contour i with one cell labelled id for each id$\in$ID such that (id,i) has a value or is marked "declared". (For example, the 1 column of the storage in S_4 with (p,1) having a value maps to the contour 1 in S_4' with a cell for p.)
2) if there is no $\aleph$ in this contour i then the contour is that of a block entry. The static link of contour i points to contour i-1.

3) if there is ✱ in this contour the contour is that of a procedure call. In (✱,i) of the storage of S is a label which identifies the statement after the call which resulted in this procedure activation. The top text of the stack which has been saved in the label value will have some procedure-valued unique name (p,j) in the statement immediately preceding that pointed to by the label's ip. In the procedure value stored in (p,j) there is a single modified text of the program. The procedure's ip points to the entry point of the procedure. That body is immediately nested in some modified block or procedure which is numbered k. That is the procedure's ep; so the static link of contour i points to contour k. (For example, consider column 2 of the storage of the snapshot taken just after a procedure call in section 5.4. (✱,2) contains the label whose ip is just after the call $p_1(2)$. In (p,1) we see that the procedure body is nested immediately in a block numbered 1. So in this case contour 2 (of the call) has a static link pointing to contour 1.)
4) store into each cell id of the contour i the value converted from the contents of (id,i). The conversion is performed according to data type:
a) integer, real and boolean values are copied as is.
b) pointer values (id',j') are converted to pointers pointing to the cell for id' in contour j'.
c) label values consisting of an ip and a stack s are converted to ip' and ep' by the following: take ip' as a copy of ip. In the text on top of s the ip points to a statement immediately nested inside a modified block or procedure numbered k. The ep' points to the contour k.
d) procedure values consisting of an ip and a text t are converted to ip' and ep' by the following: Take ip' as a copy of ip. In the text t the ip points to a statement immediately nested inside a modified block or procedure numbered k. The ep' points to the contour k.
e) "undeclared" marks are stored as blanks.

B. The ip, the stack and the BECG of S are used to form the processor of S'. More specifically, the ip' and ep' of the processor are formed by following step A4c (label conversion) with the ip of S and the stack of S as s. Then the BECG of the processor is a copy of the BECG of S.

Secondly, we informally describe the mapping $\psi(S')=S$.

A. We form the stack of S by using the contours and the processor of S'. The stack consists of texts of programs which contain modified texts of each activation of a block or procedure that has not been exited. The first step is to form the dynamic chain list by starting with the contour pointed to by the processor's ep and tracing down the chain of dynamic predecessors.
1) let C be the contour pointed to by processor's ep.
a) add the number of C to the list
b) find the dynamic predecessor C_p of C: If C is that of a block entry (no ✱ cell in C) C_p is the contour pointed to by the static link of C. If C is that of a procedure entry (an ✱ cell in C) C_p is the contour pointed to by the ep of the label stored in the cell for ✱ in C.
c) if the dynamic predecessor C_p of C exists let C be C_p and go to (a); otherwise halt.
(For example, the dynamic chain of S_4' is(1).)
2) reverse the order of the dynamic chain list, creating list $D=D_1,\ldots,D_n$.
3) now create the stack from bottom up. The bottom of the stack, i.e. $Stack_0$, is a copy of the algorithm of S'. Now for i, $1\le i\le n$, do a-b:
a) let b be the block whose entry caused creation of contour number D_i.
b) $Stack_i$ is obtained by copying $Stack_{i-1}$ and then subscripting with D_i each occurrence, in block b, of each identifier declared in b, and then adding to the begin delimiter of b the number D_i.

B. The ip of S is a copy of the ip' of the processor of S'.

C. The BECG of S is a copy of the BECG' of the processor of S'. (From S_4' we get D=(1). Hence $Stack_0$ is the algorithm of S_4' and $Stack_1$ is obtained from $Stack_0$ by subscripting each occurrence of p by $D_1=1$ and following the open parentheses of the outer block by 1.)

D. We form the storage component of S by using the contours of S'. For i, $1\le i<$BECG-1, do:
for each cell labelled id in the contour i its contents are converted for storing into (id,i) of S according to the data type:
a) integer, real and boolean values are copied as is.
b) a pointer pointing to the cell for id' in contour j is converted to (id',j').
c) label values consisting of ip' and ep' are converted to an ip and a stack: Take ip as a copy of ip'. Form the stack by following step A with C initially as the contour pointed to by ep'.
d) procedure values consisting of ip' and ep' are converted to an ip and a modified program text: Take ip as a copy of ip'. Form a stack by following step A with C initially as the contour pointed to by ep'. Then take as text the top of this stack.
e) blank cells are converted to "declared" marks.

Thirdly, we show that ϕ and ψ meet the requisite conditions.

A. Clearly $\phi(S_0)=S_0'$ and $\psi(S_0')=S_0$, for the empty environment in S_0' maps to the uninitialized storage component in S_0 and vice versa.

B. If S_i is final then $\phi(S_i)$ is final and if S_i' is final then $\psi(S_i')$ is final. In both models the final snapshot is indicated by a null ip. A null ip maps to a null ip under both mappings.

C. $Output_{CR}(S_i)=output_{CM'}\phi(S_i)$ and $output_{CM'}(S_i')=output_{CR}\psi(S_i')$. Simply let output component in CR be the entire initialized storage component and the output component in CM' be the entire record of execution.

D. 1) if $\phi(S_i)=S_i'$ and S_i is not final then $\phi(S_{i+1})=S_{i+1}'$.
2) if $\psi(S_i')=S_i$ and S_i' is not final then $\psi(S_{i+1}')=S_{i+1}$.

We consider (1) above. To get from S_i to S_{i+1} and from S_i' to S_{i+1}' one statement is executed in each model. Since ϕ maps the ip of S_i to an ip of the processor of S_i' which is pointing to the same statement, the same statement is executed in both models. Therefore we must exhaustively consider each statement type.

Block and procedure entry: In S_i and S_i' the BECG are equal; say it has the value k. Then the block is entered. A new text is pushed onto the stack with the block that was entered numbered with k and its identifiers subscripted with k. This maps under ϕ to the processor's ep pointing to a new contour k and the declared identifier's being in contour k. The ip's, of course, are still the same.

Block exit: In S_i and S_i' we have that, in the top-of-stack text the number at the beginning of the block being exited is equal to the number of the contour to which the processor's ep points; let this number be k. Then the top text is popped. We are now executing in a new top-of-stack text in a block numbered $k-1$ (this is for a block exit only). This maps under ϕ to the processor's ep pointing to the contour $k-1$ which is necessarily pointed to by the static link of contour k.

Procedure exit and goto: The label in S_i, consisting of an ip and a stack, maps under ϕ to an ep' and ip' in S_i'. The goto results in replacing the main ip and stack of S_i by that of the label. In S_{i+1} the new ip and stack map under ϕ to a processor which consists of the ep' and ip' which would be obtained from the label.

Assignments: Since in S_i the values that are assigned are either copied from already properly mapped values or are constructed from an ip and a stack which already maps to an ip' and ep' in S_i, the snapshot S_{i+1} maps under ϕ to S_{i+1}'.

Similar reasoning is used to show that the mapping ψ is preserved by statement execution. This completes the proof.

We now complete the equivalence.

Theorem 2: CR is equivalent to CM for a block structured language L.

We have shown that one of the formal definitions of block structure, CR, does imply retention rather than deletion. Other formal definitions exhibit the same preference for retention. Definitions based on the lambda calculus [Lan 65] have the potential of defining retention, because lambda calculus requires interpreters which have retention (see footnote, section 4). Various Vienna Definition Language definitions of block structured languages are easily shown to have retention. These include the definition of EPL (Example Programming Language) in the report defining the Vienna method [LLS 68] and the definition of Algol 60 [Lau 68].

8. THE STACK MODEL AND THE COPY RULE

If the copy rule is equivalent to the contour model, where does the stack model fit in the scheme of things? Henhapl and Jones [HJ 70] answer this question partly. They use their version of the copy rule in a proof that the copy rule is equivalent to the stack model for block structured languages in which there are no pointer values and in which labels and procedures are restricted to being constants or parameters of procedures. These restrictions exactly mirror those of Algol 60. Therefore, the deletion strategy may be used safely only for restricted block structured languages.

9. CONCLUSION

This paper has examined two different block and procedure exit strategies, deletion and retention. To explore the consequences of these two strategies, we defined two information structure models, SM and CM, each of which implements one of the strategies. By comparing SM and CM it was shown that at the very least retention was cleaner and more aesthetic. To resolve the question of correct strategy a formal definition of block structuring was given in the form of CR. CR was proved to be equivalent to CM with no restrictions on the language. Furthermore, others have proved that the stack model is equivalent to the copy rule for only restricted languages. Thus retention was shown to be the formally correct strategy and that which required fewer restrictions.

These results indicate that the trend towards the deletion strategy should be re-examined. It clearly behooves language designers to consider the problems of designing languages which follow the more general retention strategy. (An exposition of these problems is outside the scope of this paper. Interested readers are urged to consider [Joh 71], [Weg 71] and [Bry 71] and the references cited therein.)

Finally, it is hoped that this paper illustrates how formal techniques involving information structure models may be used to formally resolve language design issues.

Acknowledgements. The author would like to thank Katrina Avery, John Johnston, Peter Lucas, Clem McGowan, Jerry Rubin and Peter Wegner for their encouragement, time and help in the preparation of this paper.

BIBLIOGRAPHY

Note: PASODSIPL is Proceedings of ACM Symposium on Data Structures in Programming Languages, 1971.

Bee 70 Beech, D., "A structural view of PL/1", Computing Surveys 2:1,33 (March 1970).

Brg 70 Bergeron, R.D., Gannon, J.D., Tompa, F.W., Shecter, D.F., and van Dam, A., "Systems programming languages", Advances in Computers 11 (1971).

Bry 71 Berry, D.M., "Introduction to Oregano", PASODSIPL, SIGPLAN Notices (Feb. 1971).

Chu 41 Church, A., The calculi of lambda-conversion, Princeton, 1941.

Dij 60 Dijkstra, E.W., "Recursive programming", in Programming systems and languages, S. Rosen, McGraw-Hill, N.Y., 1967.

HJ 70 Henhapl, W., and Jones, C.B., The block concept and some possible implementations, with proofs of equivalence, IBM Laboratory Vienna, Tech. Rep. TR 25.104 (1970).

Joh 71 Johnston, J.B., "The contour model of block structured processes", PASODSIPL, SIGPLAN Notices (Feb. 1971).

Lan 64 Landin, P., "The mechanical evaluation of expressions", Computer Journal 6:4 (1964).

Lan 65 Landin, P., "A correspondence between ALGOL 60 and Church's lambda-notation", CACM 8: 2 & 3 (1965).

Lau 68 Lauer, P., Formal definition of Algol 60, IBM Laboratory Vienna, Tech. Rep. TR 25.088 (1968).

Luc 68 Lucas, P., Two constructive realizations of the block concept and their equivalence, IBM Laboratory Vienna, Tech. Rep. TR 25.085 (1968).

LLS 68 Lucas, P., et al., Method and notation for the formal definition of programming languages, IBM Laboratory Vienna, Tech. Rep. TR 25.087 (1968).

LW 69 Lucas, P., and Walk, K., "On the formal description of PL/1", Annual Review in Automatic Programming 6:3, 105 (1969).

McG 70a McGowan, C., "The correctness of a modified SECD machine", Second ACM Symposium on Theory of Computing (1970).

McG 70b McGowan, C., "An inductive proof technique for interpreter correctness", Courant Institute Symposium on Formal Semantics of Programming Languages (1970).

McG 71 McGowan, C., Correctness results for lambda calculus interpreters, Ph.D. thesis, Cornell University, 1971.

MW 71 McGowan, C., and Wegner, P., "The equivalence of sequential and associative information structure models", PASODSIPL, SIGPLAN Notices (Feb. 1971).

Nau 60 Naur, P., "Report on the algorithmic language Algol 60", CACM 3:5 (May 1960).

Nau 63 Naur, P., "Revised report on the algorithmic language Algol 60", CACM 6:1 (Jan. 1963).

RR 64 Randell, B., and Russell, L.J., Algol 60 implementation, Academic Press, New York (1964).

Rub 71 Rubin, G., A contour model lambda calculus machine, TR 70-35, Center for Computer and Information Science, Brown University (Feb. 1971).

vWn 69 van Wijngaarden, A., et al., "Report on the algorithmic language Algol 68", Num. Math. 14, 79-218 (1969).

Weg 68 Wegner, P., Programming languages, information structures and machine organization, McGraw-Hill, New York (1968).

Weg 70a Wegner, P., "Three computer cultures: computer technology, computer mathematics and computer science", Advances in Computers 10 (1970).

Weg 70b Wegner, P., "Information structure models for programming languages", Courant Institute Symposium on Formal Semantics of Programming Languages, 1970.

Weg 70c Wegner, P., The Vienna Definition Language, TR 70-21-2, Center for Computer and Information Science, Brown University (1970).

Luc 70 Lucas, P., lecture on Proofs of correctness of various block structure implementations, given at Brown University, spring 1970.