

Variable Access in Languages with Higher Order Procedures

CS/79/2

A.Davie

Department of Computational Science,
University of St Andrews,
North Haugh,
St Andrews,
Fife KY16 9SS,
Scotland

Summary

The usual way of implementing a block or procedure structured language is by use of a display which gives access to the various static levels of variable declaration on a stack at run time. In addition a dynamic record of return information has to be maintained.

This method makes the assumption that storage will be released in the reverse order to that in which it was claimed (i.e. in a stack-like way). Many languages, like Pascal and Algol 68, have the facility to claim storage from an alternative source called the free storage area or heap. The stack mechanism is not powerful enough to cope with all situations: for instance in a language which has pointer variables which can survive after the objects they point to have gone out of scope; or in a language which can have higher order functions with non-local variables.

An alternative is proposed based on Landin's concept of a closure in which only two levels of storage are recognised by the run-time system in a block or procedure - the local variables and the non-local variables. The user, however, can still declare variables at any lexicographic level. Access to variables is very simple at run time through two vectors of identifiers. The method has been used in the implementation of a language called IDEA.

KEYWORDS: Heap storage, Block entry and exit, Procedure entry and exit, Storage management, Higher Order Functions, Retention, Variable Binding

CR Categories: 4.10, 4.12, 4.20, 4.22

1. Introduction

Many high level structured programming languages (e.g. Algol W[27], Pascal[28], Algol 68[26] and Simula[3]) give users access to two kinds of storage - local items kept on a stack and global items kept on a heap. The former is administered on a last-in-first-out basis which seemed ideal to implement the earlier block structured languages(e.g. Algol 60[14] and Euler[29]) where the *scope* of variables (i.e. the lexicographic area of program where they were accessible) matched their *extent* (or *lifetime*). As a block or procedure was entered, storage on the stack was set aside for locally declared variables and at exit this area was relinquished by retracting the stack pointer.

In most languages where heap storage may be used, the user must specify this specifically by using a keyword like **heap** (Algol 68) or **new** (Pascal) or implicitly by using *records* (Algol W) or *class* instances (Simula). The user has to make a conscious decision about the extent or lifetime of his variables and the two different kinds of storage are subject to different sets of rules. For instance, in Algol 68, a local variable, *a*, of mode *m* may not be referred to by variables of mode **ref** *m* if they are declared in a range (block) outer to the one in which *a* is declared. Such languages, besides giving the user more to think about, are still, in most cases, implemented using a stack and objects on the heap are usually addressed via the stack. This is for two reasons. Firstly the names used to access heap objects still have static scope, only being available for use in the block in which they are declared. Secondly, it is convenient for a garbage collector to find all its base pointers on the stack and from these to mark the heap space still in use. Such stacks are further complicated by the organisation of two linked lists called the static and dynamic chains and often by the use of a vector called the *display* vector. These are described in more detail in the next section.

In the proposal set forth here, the stack is still used for the dynamic chain but the static chain and display are replaced by two *access descriptors* which make declared variables available. At each dynamically entered block or procedure there is one for local variables and one for non-locals - also known as free variables. (Note that this is an implementation detail and makes no difference to what the programmer sees — several levels of

block and procedure with variables declared at any level.) This mechanism is described in detail in later sections.

The initial reason for using this method is that any method using a stack discipline alone cannot cope with certain cases where variables are retained for use after the block in which they were declared has been left. This can arise in two different ways. One is well known (the "dangling reference" problem) and involves a pointer from an outer block pointing at a variable in an inner one. This is specifically forbidden in Algol 68 and in other languages their framework is such that the problem cannot arise. The second problem arises when procedures are given "first-class citizenship". In Algol 60, procedures could be declared, called and passed as parameters. In Algol 68, with its concept of orthogonality, an attempt was made to raise the status of procedures and allow them, like any other object, to be assigned, to be members of arrays or structures and to be returned as results from other procedures. Higher order procedures and functions were to be allowed. However this freedom is (to quote Reynolds [18]) "subject to certain 'scope' restrictions (which are imposed to preserve a stack discipline for the storage allocation of the representations of functions and labels.)". Procedures with non-local variables cannot be passed out to positions where any of those variables are out of scope. Rayward-Smith [17] says that he "...has found no elegant way of dealing with this problem and has come to the reluctant conclusion that scoping [in Algol 68] so severely limits the usefulness of the facility which enables procedures to be regarded as values that it is nearly rendered useless."

For example :-

```
begin
  proc adder = ( real a ) proc ( real ) real :
    ( proc ( real b ) real : ( a+b ) );
  proc ( real ) real successor, plus3;
  successor := adder(1);
  plus3 := adder(3);
  print (( successor(5), plus3(2) )
end
```

Here an attempt has been made in Algol 68 to define a procedure `adder` and use it in two subsequent assignments but they are illegal because the body of `adder` is another procedure with a non-local variable `a` which lives on after `adder` has been left. The Algol 68 revised report[26] makes it "clear" that use of such procedures in this way is illegal. It is embarrassing

that such usage cannot be checked for at compile time and is difficult to check at run time.

The example given above is a rather trivial one and it may not be readily obvious what use higher order functions are. A short discussion of some uses to which they have been put — and some examples — is given in the appendix.

The retention of non-local variables that live on in these ways cannot be coped with by the stack mechanism because by the time the retained non-local variables are needed, the stack has retracted and the space is either no longer in use or being used by some other block.

In this paper a method is presented where the variables can be found by a simple offset from one of two base pointers (with a possible single indirection after that.)

The method has been used in an implementation of a language called IDEA[6] which is an extension of Wirth's language Euler [29]. In IDEA, the types of scope restriction present in Algol 68 and BCPL are not present and procedures can be of higher order and can be stored, assigned, passed back as results and in general have first class citizenship. BCPL was used to write the IDEA compiler and interpreter.

1.1 Related Work

It is interesting to note that BCPL [19] makes functions and routines into first class citizens — but only at the expense of a different (and more severe) kind of scope limitation. Variables used in functions and routines either have to be local or defined as global to the whole program by placing them in a special GLOBAL vector.

LISP [10] allows function producing functions in which the static scope problems mentioned here are solved (via the FUNARG mechanism [13,24]). However the values of variables are found by an expensive run-time look-up of an association list.

However Sussman and Steele [22] describe the implementation of a dialect of LISP called SCHEME which obeys static scope rules. Steele [20] points out that this allows efficient variable access without the complications of association lists because all the binding can be done at compile time. He

presents there an optimizing compiler for SCHEME which allows variable access in a number of ways including stack and heap based methods.

In [11], Marlin describes a version of Pascal in which activation records are on the heap. He points out that this is necessary for implementation of coroutines. It is interesting to note that coroutines can be implemented in a language with properly implemented higher order procedures. When a routine is to be suspended it must pass back a procedure as its result which can be called to reactivate it.

Oregano [1], designed by D.M.Berry, solves the retention problems in a slightly different way from that described here. This is discussed in a little more detail in the current paper's section — "Discussion and Conclusions"

1.2 Layout of the rest of the Paper

Both of the problems described above, involving retention of variables after their declaring block or procedure has been left, disappear if the method described here is used. After a section giving a summary of the "traditional" methods of display plus stack for variable access, details of the proposed method are given in the three sections — 3. "An alternative Method", 4. "Block Entry and Exit" and 5. "Procedures". The penultimate section consists of an example of a complete program's execution and the last one contains some discussion and conclusions.

2. The Traditional Stack and Display Mechanism

Many implementations use a mechanism which has been called a *display* to access named variables sitting on a stack (Dijkstra[4], Randell and Russell[16]). An extremely lucid exposition of this whole mechanism is given in [8] by Johnston, but a summary is given here.

The stack is divided into *frames* and a new frame is added to the top of the stack whenever a new block or procedure is entered: and the frame is removed at exit. The frame on top contains space for the locally named variables and working space for the evaluation of commands and expressions in the block or procedure. Stack frames lower down the stack were placed there when outer blocks were entered and hence the addressability of variables is achieved by looking at the correct position inside the correct stack frame.

The stack frame also contains book-keeping information sometimes called *mark stack control words* (Hauck & Dent[7]) This block of words contains links to previously activated frames. One is a link to the frame immediately lower down the stack which is the one corresponding to the previously activated block or procedure and is there so that a safe return to the dynamically enclosing unit can be made. This is known as the *dynamic link*

and such links form a list known as the *dynamic chain*. The mark stack control word also contains information about the lexicographically enclosing block or procedure and there is a *static chain* of pointers connecting the frames of all the procedures or blocks which have currently accessible variables in them. Since we wish to access these variables without having to follow this chain to the required stack frame, it is advantageous to keep a copy of the values of the pointers in the static chain in a sequential vector called a *display*. Accessible items on the stack can then be identified by two numbers — the lexicographical block level, b , and the address of the item relative to the base of the stack frame, a . The absolute address is then $display[b]+a$. Further speed of access to variables may be achieved if the display vector is kept in fast registers or in special hardware.

Maintenance of the display, especially at procedure exit time, requires some complex manipulation. Morrison[12] gives a good example and points out that “some compilers simply dump the whole display onto the stack on procedure entry and restore it again at exit”. If this rather gross solution is not followed, the display must be recovered at exit time by running down the static chain and entering the required static links in it.

It would therefore be advantageous if an alternative to the display and static chain method could be found. The approach presented here proposes just such an alternative.

3. An alternative method

As an alternative to the use of displays and the static chain, a solution is presented here where variables are not kept on the stack at all but are *all* on the heap.

The solution depends upon the fact that variables in procedures and blocks are of two kinds — local and non-local. It makes no distinction (at run-time) between non-local variables at different lexicographical levels. Access to variables is via a *block descriptor*. This consists of two pointers (possibly kept in fast registers) to vectors — one of local variables called the *local heap frame* and the other of addresses of non-local variables and called the *non-local heap frame*. This is perhaps best illustrated by an example.

Consider the following program fragment where a and b are locally declared and c and d have been declared in (perhaps different) outer blocks:-

Example 1

```

.
.
.
begin
  new a;
  new b;          comment IDEA declarations;
  .
  .
  d := a+b-c
  .
end
.
.
.

```

When the statements in this block are being executed, the block descriptor is as in Fig. 1.

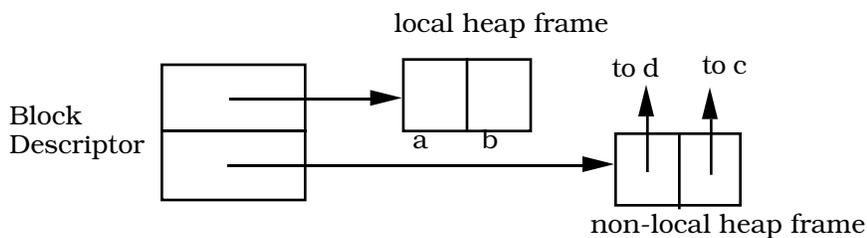


Figure 1

c and d will appear in (possibly different) local heap frames of outer blocks.

Thus we can access local variables directly by an offset from the start of the local heap frame and non-locals indirectly as offsets from the start of the non-local heap frame.

4. Block entry and exit

When an inner block (IB) is entered, three things happen.

1. The block descriptor pointers from the outer blocks (OB) are stacked for return purposes as part of the mark stack control word together with dynamic information as before. Note that the stack is still utilised for the dynamic chain and working storage.
2. A vector of space is obtained from the heap for the new non-local heap frame for IB and the addresses of any non-locals are filled into it. The OB has access to these variables and can do this. A pointer to the vector is placed in the non-local cell of the block descriptor.
3. A vector of space is allocated (again from the heap) for the new variables declared in IB and this becomes the new local heap frame. A pointer to it is placed in the local cell of the block descriptor.

Example 2

```
begin comment block A;
  new a; new b;
  .
  .
  begin comment block B;
    new c;
    .
    .
    a := c;
    .
    .
    begin comment block C;
      new e; new f; new g;
      .
      .
      b := c+e;
      .
      .
    end
  .
  .
end
.
end
```

On entry to the outer block, A, the block descriptor is as shown in Fig. 2a.

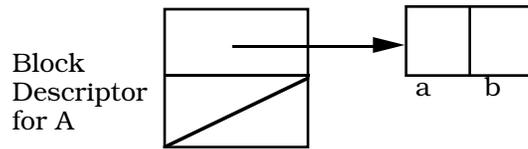


Figure 2a

At entry to block B, the above descriptor's components will have been stacked and the current block descriptor is as in Fig. 2b.

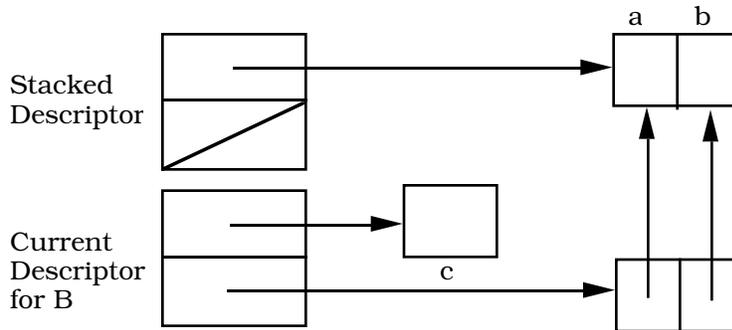
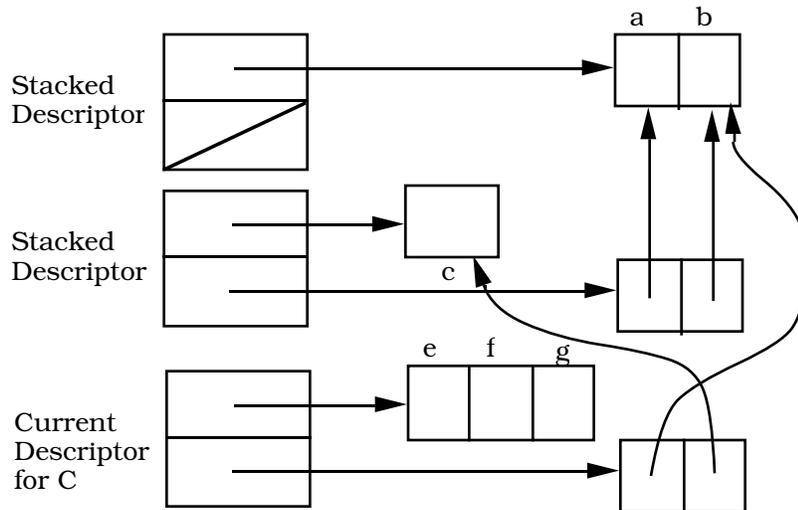


Figure 2b

Note that the address of b is needed so that it may be passed into block C when the situation is as shown in Fig. 2c.



Exit from blocks is very simple. The components of the block descriptor for the outer block are merely unstacked and the heap frames that the descriptor used to point at are left for the garbage collector.

5. Procedures

The situation with procedures is slightly more complex. We do not only have to think of the times of entry and return but also the moment at which the block in which its text appears is entered because it is at this time that the non-local variables of the procedure have to be given a corresponding internal address. That is, they have to be bound statically, not dynamically.

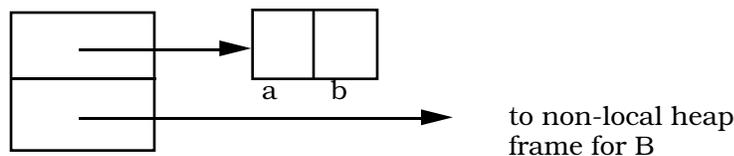
In IDEA, procedures are anonymous objects which may be considered to be constants of type **function**. They may be assigned, used as vector elements, returned as results and in general behave like any other constant and can appear anywhere that constants can. They are represented by blocks of code with declarations of formal parameters at their head and surrounded by quote marks. (For further information about anonymous procedures, consultation of Wirth's paper on Euler[29] should be made, though parameter passing in IDEA is a slight simplification of the way it is done in Euler, all of them being passed by value).

If a procedure *P* is textually enclosed in a block *D* (i.e. appears in *D*), then it is the task of *D* when *D* is entered, to manufacture what is called a *closure*. (This notation is taken from Landin[9]). If the procedure is assigned to a variable *V* then a call of *V* must know about the non-local variables of *P* and this information is part of the closure and is exactly that contained in the non-local heap frame for *P*. Thus the closure consists of a two element vector (taken from the heap). The first element contains the entry point of *P* and the second a pointer to a non-local heap frame describing the non-local variables in *P*. A pointer to this two word vector is the value that is assigned to *V* and which would be passed around if *P* were to be used in other ways besides assignment. Thus *V* carries around *P*'s non-local environment with it.

Example 3

```
.  
begin comment block B;  
  new a; new b;  
  .  
  .  
  a :=  
    '      comment procedure P between ' ` ;  
    formal c;  
    formal d;  
    .  
    .  
    c := d+b;  
    .  
    ;  
  .  
end  
.
```

If the block descriptor for B is as in Fig. 3a.



and b is the only non-local variable in P, then after the assignment to a has taken place we have

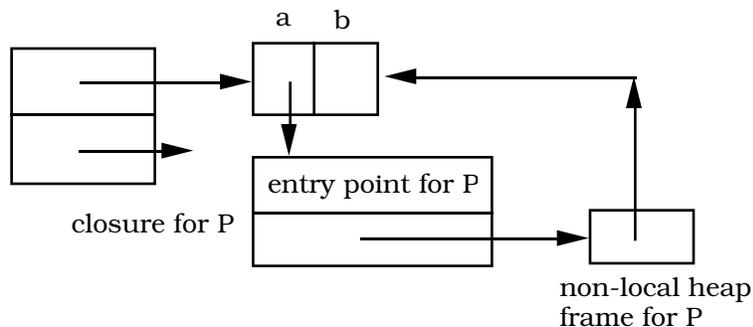


Figure 3b

Closures are manufactured (from heap space) when the procedure is declared. (Perhaps "declared" is not the correct word. The procedure is anonymous and no variable name is being declared, but the usage here is that a procedure is said to be declared when the block (Here B) in which it is embedded is entered).

The closure will be used when the procedure is called. At call time, the parameters are evaluated (if any) and a local heap frame manufactured with their values stored in them. The non-local heap frame is obtained from the closure and the block descriptor is manufactured out of these two after the current block descriptor's components are stacked together with the return address and dynamic link in the mark stack control word.

Example 4

If, in block B of example 3, procedure p is called by a statement containing

....a(3,4)....

the block descriptor is manufactured before P starts execution as shown in Fig. 4.

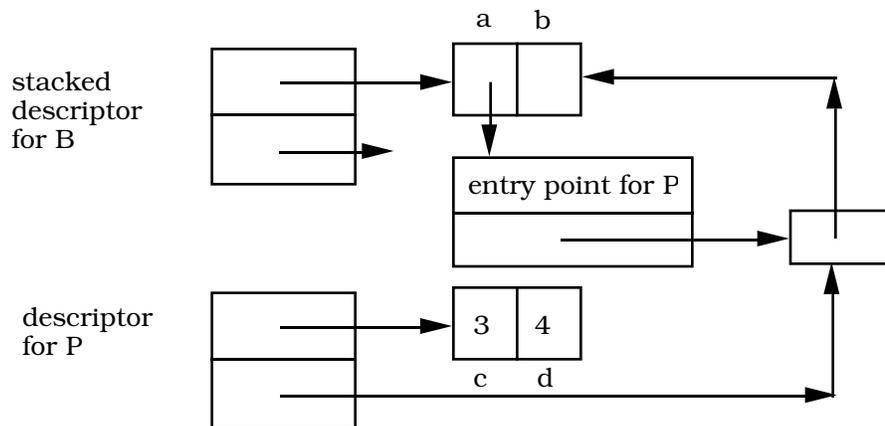


Figure 4

6. A concrete example

In this section, an example is given where a procedure produces, as its result, another one. The resulting procedure has a non-local variable that outlives its scope and therefore a purely stack based method will not work in this case.

Example 5

```
begin comment block B;
  new adder; new successor; new plus3;
  adder :=
    ' formal a; comment procedure P;
      ' formal b; comment procedure Q;
        a+b
    ,
    ,
    ,
  successor := adder(1);
  plus3 := adder(3);
  print(successor(5),plus3(2))
end
```

Example 5 presents a complete program of a rather trivial nature. Procedure P is assigned to the variable `adder`. The body of procedure P is another procedure, Q, which gets returned as the result when P is called. This happens in two places. First the procedure is called with parameter 1 and the resulting procedure assigned to `successor` which is therefore now a function for adding 1 to things. Similarly, `plus3` becomes a function for adding 3 to things; and so the final print statement should put out 6 and 5.

Let us follow the progress of execution of this program. On first entering the outer block, the block descriptor will be as shown in Fig. 5a.

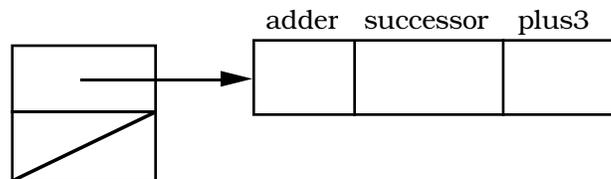


Figure 5a

Just before the first assignment, a closure for procedure P is manufactured. It has no non-local variables, so the closure assigned to `adder` is as shown in Fig. 5b.

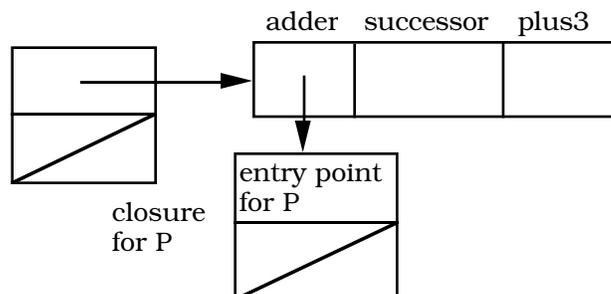


Figure 5b

Now a call is made to `adder` with parameter 1. We stack a mark stack control word consisting of the block descriptor for B and a dynamic link and return address. We then enter procedure P. The situation is as shown in Fig. 5c.

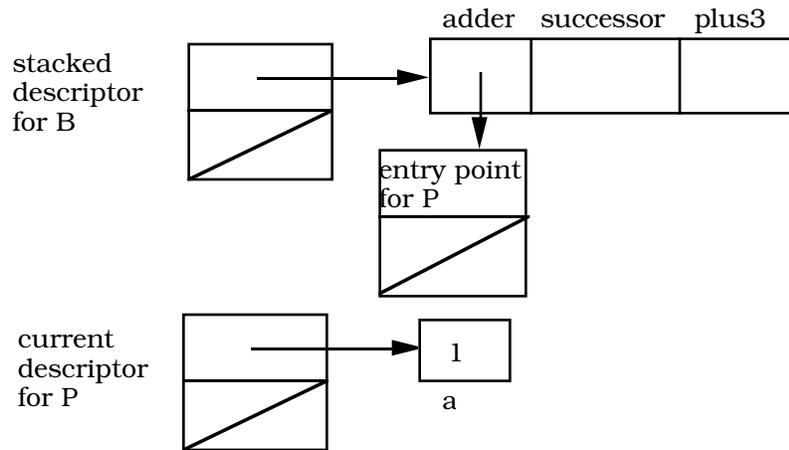


Figure 5c

The execution of the procedure P consists of manufacturing a closure for procedure Q and returning it as the result. The closure returned is as shown in Fig. 5d where the return has happened and the assignment been made. Note that this time there *is* a non local variable referred to in the closure.

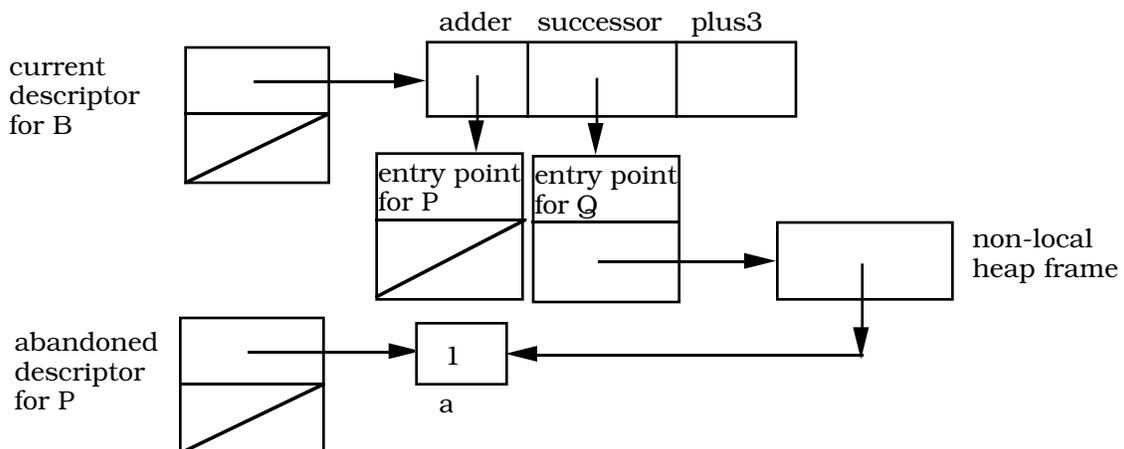


Figure 5d

A second entry to P is now made, this time with parameter 3. The picture is as given in Fig. 5e. Note that there are now two versions of the out of scope variable `a` hanging around, one in each closure.

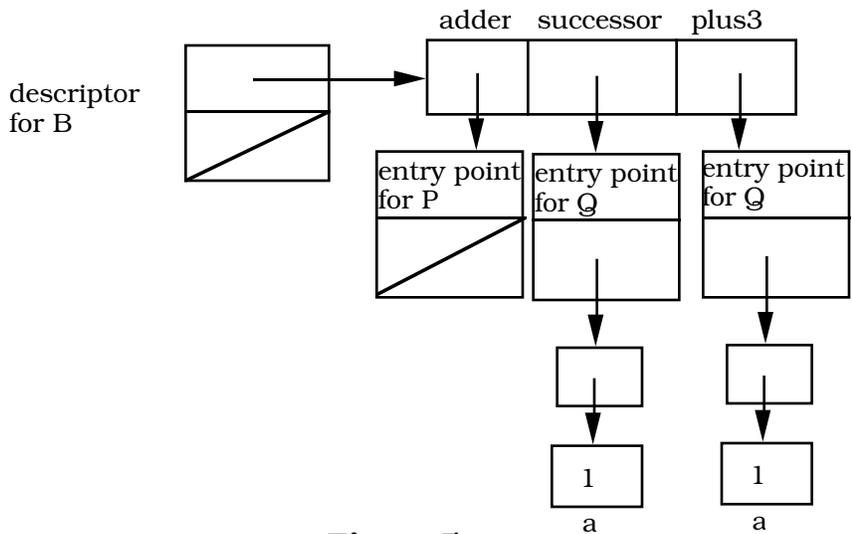


Figure 5e

Before the print is carried out, successor is entered and we get the situation shown in Fig. 5f.

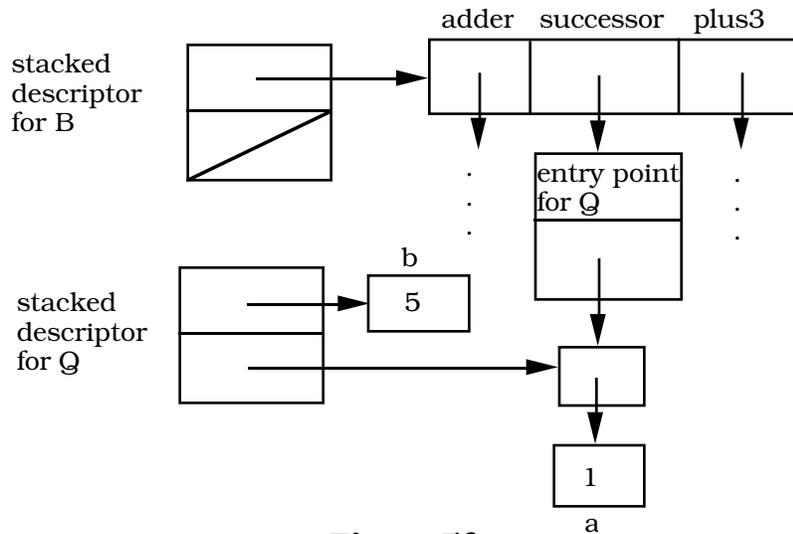


Figure 5f

We can now see that Q has access to its local parameter $b (=5)$ and to the proper version of $a (=1)$ through the non-local heap frame. The result, 6, is returned (via the stack).

Similarly, when Q is called for the second time via `add3(2)`, the proper value of $a (=3)$ will be available because the different closure is used when making the non-local heap frame.

7. Discussion and Conclusions

The method described here has some advantages and some disadvantages. On the positive side, there are three points that can be made :

(i) The user does not have to bother about whether his objects are on the heap or on the stack — All named objects are on the heap.

(ii) Orthogonality is improved. A new freedom is given to procedures and pointers. They become “first class citizens” with full “civil rights” being :-

- assignable
- passable as parameters
- returnable as results
- able to be components of vectors and structures
- nameable

There are no special rules which prevent these being used if variables outlive their scope.

(iii) If an inner block accesses some non-local variables, only those variables are retained, even though there may be other variables declared at the same level(s) as them. D.M.Berry, in his implementation of Oregon [1], has suggested a method which also solves the retention problems outlined and solved here but it would appear that in Oregon, if a non-local variable is needed, this involves the retention of the whole declarative layer (or "contour") in which it is defined — not of the non-local variable only.

Some criticisms, however, can be made:

(i) The fact that non-local variables are only available indirectly (since it is their address that is stored in the non-local heap frame, not their value) is inconvenient. With certain hardware architectures, it will obviously be of advantage to keep a copy of some or all of the current non-local heap frame in fast registers.

(ii) Although procedure exit is much easier than with the display plus stack method, there is quite a lot of bookkeeping to do on block entry. All the non-locals for the block being entered must be loaded up into the new non-local heap frame and closures may need to be manufactured. One possible solution or palliative is to use procedure

rather than block level addressing. Wichman[25] (section 1.8.2) points out that one may virtually remove the equivalent of blocks from the object program and only manage space allocation at procedure entry and exit. This cuts out the overhead at block entry and exit. One disadvantage remains. Storage may be allocated for variables in blocks (inside conditionals) which are never entered.

(iii) In any system with a heap there must be some way of releasing space for reuse once it has outlived its usefulness. This is traditionally performed by a garbage collector. Some systems like Pascal leave it to the user to return space explicitly, but the author prefers the former automatic approach because with several levels of recursion and procedure producing procedures in operation it may not be clear to the user what space *is* still in use. The last disadvantage which can be mentioned is that a viable garbage collector for the above scheme will have to be moderately sophisticated. Since any of the variables in a local heap frame can be pointed at by a cell in a non-local heap frame if it is used non-locally (see Fig. 6), each of them must be individually *markable* by a garbage collector if it is to collect cells no longer in use in the local heap frame (see shaded area).

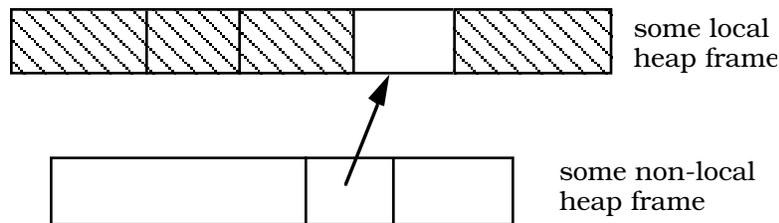


Figure 6

This implies the need for one of

- A machine architecture with spare mark bits in each addressable unit. The author knows of no such machine code facility except by emulation.
- Interpretation of such a machine architecture. This is the solution adopted in IDEA (where other bits are also used to indicate the dynamically allocated types of each object).
- Use of a separate bit map. Here an area of store is set aside at garbage collection time which contains a number of bits equal to the

number of addressable units in the heap. These bits are used for marking the corresponding units as “in use” for garbage collector purposes. This solution may be inefficient unless the implementation machine has a bit-addressable capacity.

Steele [20] has suggested that “the compiler has complete discretion over the manipulation of environments and variable names.” His work on an optimizing compiler for SCHEME [22] shows that certain environments may be stack allocated if this is felt to be more efficient.

He also says that “the form of the environment is completely arbitrary as far as the SCHEME interpreter is concerned; ... this cannot matter since the only code which will ever be able to access that environment is the code belonging to the functional closure of which that environment is a part.” But this is not entirely the case. Parts of environments may be *shared* by two different closures; and it is no good merely making a *copy* of the associated storage cells as such “aliases” may change the value of the cell by assignment (unless the language being implemented is purely applicative) and each alias must have access to the updated version of the cell. This is the reason that in the proposal presented here the addresses of the non-locals are kept in the descriptor, not their values. If it could be determined that a particular variable had no aliases (and this is possible for SCHEME but is not in general computable at compile time for languages with pointers like IDEA) then a copy could be taken.

It should be pointed out that the method outlined here can be used even when the retention problems do not arise — i.e. in the “traditional” Algol-like languages. The variables would then be able to be on the stack again: we would replace the heap frames by stack frames, but instead of the display mechanism, a block descriptor would be used in the way outlined here. But the author feels that the main merit of the method is that it can be used to remove some of the restrictions present in more modern languages like Algol 68 and Pascal.

8. Acknowledgements

The author is grateful to Ian Hards, Dougald Gray, Ewan Murray and Alan Milne who implemented the ideas presented here as a final year honours undergraduate project and whose initials form the acronym IDEA.

APPENDIX - Uses of Higher Order Procedures

First, let us consider privacy of variables. The modular approach to programming is frequently abused because of the lack of proper facilities for variables which are private to a routine and which can only be accessed from the outside world in a controlled way. Suppose, for instance, that a pseudo-random number generator is to be programmed. We assume that a function `next` has been chosen such that the sequence of random numbers $\dots r(i), r(i+1), \dots$ is generated by the recurrence relation $r(i+1) = \text{next}(r(i))$. The trouble occurs when one wants to protect the sequence from tampering by making the last random number available to the generator but private to it. Algol 60 nearly had this facility with its ill defined `own` variables but, among other deficiencies, there was no way to initialise them. The following IDEA function solves the problem in one way :-

```
maker :=
  ,
  formal seed;
  ,
  begin
    seed := next(seed);
    seed
  end
  ,
  ,
```

When a call `generator := maker(1234)`, say, is made, a new random number generator (a function of no variables) is returned which *retains* the seed so that each subsequent call of `generator()` will produce another random number. Furthermore by writing `gen := maker (5678)`, a different generator can be manufactured which retains its own private seed.

A similar but more involved use of private variables has been implemented in IDEA. Certain languages (e.g. BCPL,IDEA) only provide one dimensional arrays. One can implement multidimensional arrays in IDEA

by writing an array accessing function generator `newarray`. This takes as parameter a list of the bounds of the array and keeps them private. It also obtains storage for the array dynamically and keeps it private. It returns an accessing function such that , for instance, if `A := newarray([1,10,1,5])` is called, then subsequent calls of the function `A` by `A(i)(j)` will get the address of the i,j^{th} element of the 10×5 array set up by `newarray`. Furthermore it is possible to write a “slicing” function that takes `A` and a row (or column) number as a parameter and returns a new array accessing function, `B`, which gets at elements of a slice of `A`.

Secondly, consider the extension of scalar functions to make them applicable to vectors. If `f` is a function of one variable, we may wish to generate a function `g` that applies `f` to each element of a list to give a new one such that

$$g([v1,v2,\dots]) = [f(v1),f(v2),\dots]$$

This can be implemented by the higher order function `map` as follows :-

```
map :=
  ' formal f;
    ' formal v;
    begin new fv,i;
      fv := length v;
      i := 1
      while i <= length v do
        begin
          fv[i] := f(v[i]);
          i := i+1
        end
      fv
    end
  `
```

Here one only has to say `g := map(f)` to provide a new function which operates on lists. LISP users will be able to think of a multiplicity of similar functions.

The last examples of the uses of higher order procedures given here are taken from the published literature. Rayward-Smith [17] uses them in list processing to achieve “lazy evaluation”. Normally when a list is constructed from a head and a tail (`CAR` and `CDR` in LISP terminology), the head and tail are evaluated before new space is allocated and the two fields stored. In lazy evaluation, it is only when the head or tail field is *accessed* that evaluation takes place. This requires that a “formula” for carrying out the

evaluation (consisting of a closure — that is a function in a certain environment) be kept to use if and when an accession is made and these formulae are implemented by procedures with non-local variables representing the environment. It's worth looking at this paper to see the extravagant lengths to which it was necessary to go to make legal Algol 68 procedures do the job properly. The author also shows how easy it would be to write the procedures if the scoping rules which prevent retention were to be relaxed. For further details of lazy evaluation see the stream functions of Burge [2] (section 3.10) and Friedman and Wise [5].

Finally let us consider the modelling of the equations of denotational semantics. The Scott/Strachey approach to semantics of programming languages (see Tennent [23] ,Stoy[21]) involves equations using many higher order functions. Pagan [15] attempts to use Algol 68 for such modelling and runs into scope difficulties of the kind mentioned here. Reynolds [18] describes a similar use of higher order functions in interpretive definitions of semantics. IDEA can be used in the way Pagan describes without any modification.

REFERENCES

1. Berry, D.M. *Introduction to Oregano* Proceedings of a Symposium on Data Structures in Programming Languages, Gainesville, Florida, SIGPLAN Notices Vol.6 No.2 (1971)
2. Burge, W.H. *Recursive Programming Techniques* Addison Wesley (1975)
3. Dahl, O.-J., Myhrhaug, B. & Nygaard, K. *The Simula 67 Common Base Language* Norwegian Computing Centre, Forskningsveinen 1B, Oslo 3. (1968)
4. Dijkstra, E.W. *Recursive Programming* Numerische Mathematik Vol.2 (1960)
5. Friedman, D.P. and Wise, D.S. *Cons Should not Evaluate its Arguments* 3rd International Colloquium on Automata Languages and Programming, Edinburgh University Press (1976)

6. Hards, I., Gray, D., Murray, E. & Milne, A. *IDEA Language Manual* Department of Computational Science, St.Andrews University, Scotland (1977)
7. Hauck, E.A. & Dent, B.A. *Burroughs B6500/B7500 Stack Mechanism* A.F.I.P.S. Conference Proceedings, Vol. 32. (1968)
8. Johnston, J.B. *The Contour Model of Block Structured Processes* SIGPLAN Notices Vol.6 No.2 (1971)
9. Landin, P. *The Mechanical Evaluation of Expressions* Computer Journal Vol. 6 (1964)
10. McCarthy, J. et al. *LISP 1.5 Programmer's Manual* MIT Press (1962)
11. Marlin, C.D. *A Heap-based Implementation of the Programming Language Pascal* Software Vol.9 No.2 (1979)
12. Morrison, R. *A Method of Implementing Procedure Entry and Exit in Block Structured High Level Language* Software Vol. 7, No. 4 (1977)
13. Moses, J. *The Function of FUNCTION in LISP* Project MAC Report AI-199 MAC-M-428 (1970)
14. Naur, P. (ed.) *Revised Report on the Algorithmic Language ALGOL 60* Communications of the A.C.M., Vol. 6, No. 1 (1963)
15. Pagan, F.G. *Algol 68 as a Metalanguage for Denotational Semantics* Computer Journal Vol.22 (1979)
16. Randell, B. & Russell, L.J. *ALGOL 60 Implementation* Academic Press (1964)
17. Rayward-Smith, V.J. *Using Procedures in List Processing* Proceedings of the Strathclyde Algol 68 Conference, SIGPLAN Notices Vol.12 No.6 (1977)
18. Reynolds, J.C. *Definitional Interpreters for Higher-Order Programming Languages* Proceedings of the 25th ACM National Conference (1972)

19. Richards, M. *BCPL: a Tool for Compiler Writing and System Programming* AFIPS Conference Proceedings Vol. 34 (1969)
20. Steele, G.L.Jr. *RABBIT: A Compiler for SCHEME* MIT Report (1978)
21. Stoy, J.E. *Denotational Semantics* MIT Press (1977)
22. Sussman, G.L. & Steele, G.L.Jr. *SCHEME: An Interpreter for Extended Lambda Calculus* MIT AI Memo 349 (1975)
23. Tennent, R.D. *The Denotational Semantics of Programming Languages* CACM Vol.19 (1976)
24. Weizenbaum, J. *The FUNARG problem explained* Unpublished Memo. MIT (1968)
25. Wichman, B. *Algol 60 Compilation and Assessment* Academic Press (1973)
26. van Wijngaarden, A. et al. *Revised Report on the Algorithmic Language ALGOL 68* Acta Informatica, Vol 5, Fasc 1-3 (1975)
27. Wirth, N. & Hoare, C.A.R. *A contribution to the development of ALGOL* Communications of the A.C.M., Vol. 9, No. 6 (1966)
28. Wirth, N. *The Programming Language Pascal* Acta Informatica, Vol 1, Fasc 1 (1971)
29. Wirth, N. & Weber, H. *EULER : A Generalisation of ALGOL and its Formal Definition : Part II* Communications of the A.C.M., Vol. 9, No. 6 (1966)