Recursive Nested 구조를 위한 Run-time 기억장소 운영에 관한 연구

論 文
31~4~1

A Study on the Run-time Storage Management for Recursive and Nested Structure

金 榮 澤* · 車 允 卿**
(Yung-Tack Kim · Youn-Kyung Cha)

Abstract

PASCAL has a recursive nested structure and uses deep binding of identifiers. This paper studies the problems and techniques in storage management for PASCAL on the IBM 370 system, and presents run-time storage administration algorithms which use stack scheme and heap efficiently on the view of storage. The stack-scheme was used to implement the feature of recursive nested structure and the heap was used to implement the feature of the dynamic allocation procedure and pointer variable, allowing an additional dynamic storage recovery procedure.

1. Introduction

PASCAL is a widely used language, primarily in university computer science departments. Complete descriptions of the language PASCAL can be found in Jensen and Wirth[9] [1975], Wirth[13] and Wirth [14], etc.

Computer science department of S.N.U. has tried to implement PASCAL compiler. This paper concentrates on the storage management in the PASCAL implementation which has recursive nested structure. Storage management is one of the first concerns of the language implementation. The rules that define the scope and duration of names in a programming language determine the strategies that can be used to allocate stroage. A more complex strategy than static allocation scheme is required for PASCAL, in which a name in a recursive procedure may have more than one activation at a given moment at run time. For languages with recursion, a dynamic allocation scheme involving a stack is usually used. Gries[7] treated stack-based storage management in some detail.

However, the stack is not a sufficient model because the stack management requires that storage be released at known moments in the opposite order from its allocation and in a completely nested manner. Languages like PASCAL, which allow the use of dynamic allocation procedure and the manipulation of pointers, do not follow this rule. PASCAL requires allocation of memory as the result of the execution of dynamic allocation procedures, and the allocation lasts as long as same live pointer refers the element of memory. For this feature,

* 正會員 : 서울大 工大 電算機工學科 敎授 · 工博
** 正會員 : 서울大 大學院 計算統計學科
接受日字 : 1981年 11月 23日

storage can be allocated from an area called a heap. Knuth[10] analyzed a number of techniques for the heap storage management.

In this paper the various problems and techniques in storage management for PASCAL, which has recursive nested structure and uses binding of identifiers to names under the most closely nested rule, are discussed on the view of the efficiency of a system. And the techniques determined for our implementation, based on the discussion, will be shown.

2. Stack Scheme for Recursive Procedure

PASCAL has a nested procedure structure and allows recursive procedure calls.

Therefore, PASCAL requires some run-time storage administration. It, however, can be implemented by a stack scheme except pointer variables which are one of the main causes of increased storage allocation complexity. In addition to the handling of recursive procedures, the last-in and first-out operation mode of a stack has the further advantage for PASCAL with nested structure, because it results in an automatic overlay of data belong to non-activated procedures. The stack management technique for a variety of control structures is discussed by Bobrow and Wegbreit[4].

The primary unit of program structure in PASCAL is the procedure (or function). One dynamic data area called an activation record is associated with each execution of a procedure for its fixed-length variables. When a procedure is activated, it takes storage for its activation record from the current top of the stack, and when it returns to the calling procedure, it pops the stack, releasing the activation record.

Since activation records allocated in the stack may have different lengths, they must be linked by particular combination of pointers called dynamic link to implement the stack allocation of storage. The dynamic link pointers chain activation records in the order of their dynamic generation. The dynamic link was implemented by locating the pointers in the particular position of the activation records. The formation of dynamic link after the second activation of Q, in example 1, is shown in Figure 1.

Example 1:

```
PROGRAM P;
  PROCEDURE Q;
      PROCEDURE R;
          CALL Q;
      END R;
      CALL R;
  END Q;
  PQOCEDURE S;
      CALL Q;
  END S;
  CALL S;
END P;
```

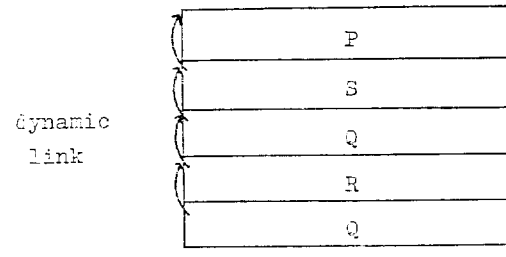

Fig. 1. Dynamic link of activation records

3. Various Display Managements for Deep Binding

PASCAL uses binding of identifiers to names under the most closely nested rule called deep binding. Thus other activation records as well as the current one must be accessible since a procedure may refer variables in enclosing procedures. If more than one activation of the enclosing procedure exist, then the topmost activation record is the accessible one.

One method useful for the implementation of the languages using deep binding is the display. The lidea of the display is from Dijkstra[6] and it has been developed in Randell and Russel[12] and Gries, Paul, and Wiehle[8]

There can be several display maintenance methods

depending on the places in which displays are stored. Four available methods are studied and the comparison of the methods is done on the view of efficiency and space. The comparison is restricted within the display management, and the management of stack and dynamic link, which is common to all methods, is omitted.

i) Static-linked Displays

The displays might be kept on the run-time stack itself. One of the methods is the static-linked displays. In static link scheme, each activation record has a static link to the activation record of its immediately enclosing procedure. In Figure 2, the static link of example 1 after call of R is shown.

The static link method allows simple entry and exit procedures. On entry the dynamic link is searched for the procedure whose level is the level of called procedure minus 1 and the static link is extended. On exit it is necessary only to delete the top activation record from the stack.

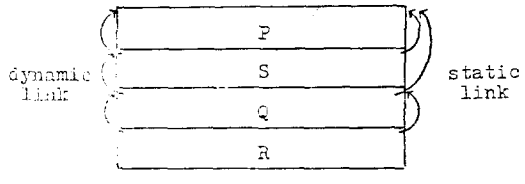

Fig. 2. Static link and dynamic link

Now the address calculation of variables in this method, which is the most crucial factor for the system efficiency, is considered. At translation time it is assumed that the run-time location of any variable is represented by an ordered pair (level number, offset) for all method. The addressing of variable in this method must execute indirect instructions [level of the current active procedure—level number of the variable] times to calculate the base address of the activation record to which the variable refers. And the offset is added to the base address to get the absolute address of the variable at code generation time. The space for display management in this method is only one for each activation record to store the static link.

ii) Static Link and A Separate Active Display

The necessity of searching the static link for each address calculation of a value decreases the efficiency of addressing considerably. We might avoid this drawback by using an alternative method in which the active display is stored in memory as a separate stack. The compiler can determine the maximum length of the stack because it is the maximum nesting depth of procedures in the program. So the additional required storage for active display can be determined.

In this method all address calculation of variables of accessible activatin records would begin by using indirect addressing through the active display pointer. The base address of the appropriate activation record of level number k is acquired by an indirect instruction;

Base address=k [Active display].

And the absolute address is calculated by the addition of the base address and offset.

Although address calculation of this method of this method is simplified, the cost of procedure entry and exit is increased for the maintenance of the active display. On procedure entry when the level of called procedure is L_2, the L_2 [Active display] must be updated to contain the value of the base address of the new top activation record. On procedure exit, the active display is updated to contain the display of a calling procedure. The static link of the called procedure, therefore, must be searched and restored at Active display. The state of the dynamic link, static link, and active display of example 2 before and after call of Q is shown in Figure 3.

Example 2:

```
PROGRAM P;
    PROCEDURE Q;
    END Q;
    PROCEDURE R;
        PROCEDURE S;
            CALL Q;
        END S;
        CALL S;
    END R;
    CALL R;
END P;
```

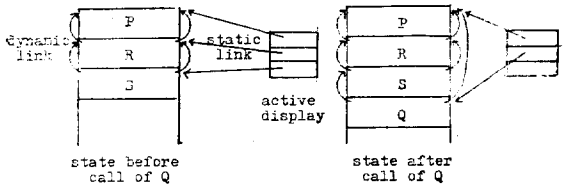

Fig. 3. Active display

iii) Local Display

Another method to store displays on the run-time stack itself is to store displays of procedures at each activation record as an array of pointers. This simplifies restoring of active display at return,although the additional space is needed for each activation record.

On procedure entry,in this method, it is necessary to copy the Display [O: L_2—1] of calling procedure at the called procedure activation record, and to define Display [L_2] for new display entry for the activated procedure when the level of called procedure is L_2. On procedure exit, no action is needed for display management.

Address calculation of variable is made in the same way as it was done in the static link and a separate active display method except using top display pointer instead of active display pointer. In example 2, the state of local displays after call of Q is shown in Figure 4.

iv) Static Link and An Active Display in Index Registers

In a computer which has a facility for indexing addresses, the available method for the efficiency of a system is to use index registers to load displays. Since index registers are usually available only in small numbers we consider the method keeping only the active display in index registers. In this method, however, we must restrict the depth of nesting to the number of available index registers.

Address calculation of variable can be acquired by a single indexed instruction;

absolute address=level number [Active display] +offset.

On procedure entry and exit, it is needed to take the same action as it was done in the static link and a separate active display method except using Active display in index registers.

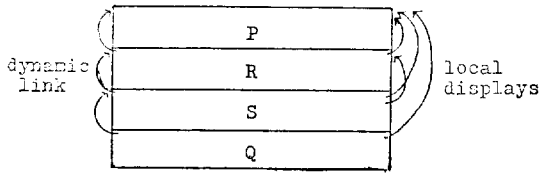

Fig. 4. Local display

4. An Implementation of Recursive Nested Structure

The entire sequence of codes executed upon procedure call, entry, and exit in our stack allocation scheme for PASCAL implementation is listed in this chapter. This scheme is based on the result of comparison of the previous chapter. It was deduced from the comparison that if a computer is available with a set of index registers, the best solution for the efficiency of a system is to reserve index registers for the active display. Our compiler is implemented for the system which has several available index registers.

Thus we restrict the nesting depth to five and use five index registers for an active display. The format of our activation record is shown in Figure 5.

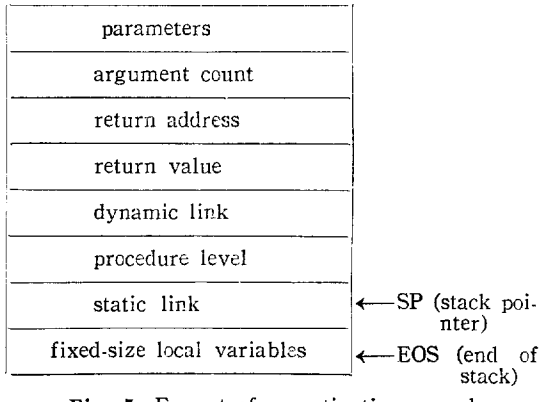

Fig. 5. Format of an activation record

The translation of an procedure call is the following quadruples:

```
        param E1, Flag1
        param E2, Flag2
          .    .    .
        param En, Flagn
          call P, n
```

where E_i is the location of an actual parameter and $Flag_i$ is 0 when the parameter is called by reference and 1 when called by value.

```
param E, 0: EOS←EOS−1
            0[EOS]←E
param E, 1: EOS←EOS−1
            0[EOS]←0[E]
```

It is assumed that memory locations are counted by words. *EOS* points to the lowest-numbered used location on the stack.

```
call P, n: EOS←EOS−1
           0[EOS]←n
           EOS←EOS−1
           0[EOS]←R    /* R is tne return address. */
           EOS←EOS−2
           0[EOS]←SP+2
           go to K     /* K is the first statement of P. */
```

The first statement of the called procedure is an quadruple 'procbegin *P*'.

```
procbegin P: EOS←EOS−1
             0[EOS]←L /* L is the level of P */
             EOS←EOS−1
             0[EOS]←D[L−1]  /* store the the static link pointer */
             SP←EOS
             D[L]←SP
             EOS←EOS−sz  /* sz is the size of P */
```

(Here $D[\phi: 4]$ is the array of index registers for the active display.)

The end of procedure is marked by an quadruple 'procend'.

```
procend: L2←1[SP]  /* L2 is the level of        */
                   /* called procedure          */
         R←4[SP]   /* R is the return address   */
         EOS←SP+5  /* EOS points to the         */
                   /* argument count            */
         EOS←EOS+0[EOS] /* restores EOS */
         SP←2[SP]−2     /* restores SP  */
         L1←1[SP]       /* L is the level of */
                        /* calling procedure */
         I←L1           /* Active display update */
         T←SP
L:       if I≥L2
           then D[I]←T
                I←I−1
                T←0[T]
                go to L
         endif
         go to R
```

6. Dynamic Allocation Procedure and Pointer Variable

PASCAL has pointer type variables and a dynamic allocation procedure which allow the allocation of memory for some data in a non-nested fashion. Therefore, PASCAL requires some run-time organization except stack scheme. A heap is a block of storage within which pieces are allocated and freed in some relatively unstructured manner. To allocate from the heap, one makes an explicit call on the standard procedure called new (Jensen and Wirth [9]).

The standard dynamic allocation procedure can be implemented by simple technique which returns heap pointer value and increases the heap pointer by the size of a requested component upon each call of the new procedure (see Figure 6)

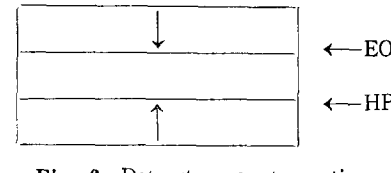

Fig. 6. Data storage at run-time

PASCAL pointer variables provide the function of the construction of complicated and flexible data structures including linked list structure. A method for returning the element to the free list is an essential part of any list processing system. The major problem of returning elements is that of knowing which part is no longer needed. Our implementation

of PASCAL allows the other dynamic allocation procedure 'dispose' defined by Addyman[2] in a draft proposal for PASCAL. Our implementation method for dynamic allocation procedures, new and dispose, is shown below.

Initial allocation of the heap is accomplished with a heap pointer and a head of the free space list stored fixed locations outside the heap (see Figure7)

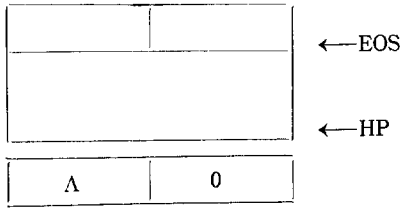

Fig. 7. Initial allocation of heap

Allocation from the heap is made from areas on the free space list whenever possible. The first word of each free storage area contains the size of that area and the address of the next free area, and the free areas are linked in order of memory address from the bottom.

If no free area list exists, or if all the areas on the list are too small, the allocation is made in the area between the EOS and HP pointers. If this area is enough, HP is increased by the number of words required and the address of the area is returned. If the area between the EOS and HP is too small, the allocation cannot be made and the procedure is terminated.

```
procedure ALLOCATE (X, N);
  /* X will have the address of the variable to be
  by this procedure. N is the size of the variable to
  be allocated and can be calculated at translation
  of new depending on the rule.                    */
        Q=addr (AVAIL);
SEARCH: P=LINK (Q);
        if P=null then  /* no free space list */
          if HP+N≤EOS then do;
            /* storage is allocated from      */
            /* between the EOS and HP         */
                        X=HP; HP=HP+N;
                        end;
                  else call NO_AVAIL_SPACE;
        if SIZE (P)≥N then go to FOUND;
        Q=P;
        go to SEARCH;
FOUND:  K=SIZE(P)−N;
        if K=0 then LINK(Q)=LINK(P)
               else SIZE (P)=K;
        X=P+K;
end ALLOCATE;
```

Recovery procedure is called by the dynamic allocation procedure dispose. Our RECOVER procedure scans the free area list. First it determines whether or not the high boundary of the storage to be freed is a tthe address pointed by the *HP* to move the *HP* for recovery of the storage used by the heap.

```
procedure RECOVER (X, N);
        Q=addr (AVAIL); R=addr (AVAIL);
SCAN:   P=LINK (Q)
        if P=null or P>X then go to HP_CHECK;
        R=Q; Q=P; go to SCAN;
HP_CHECK:
        if X+N=HP then
          do; if Q+SIZE(Q)=X then do; HP=Q;
                                LINK (R)=null
                        end;
                  else HP=−X;
                return;
            end;
HIGH BOUND CHECK:
        if X+N=P then do; N=N+SIZE (P);
                          LINK (X)=LINK (P)
                     end;
                else LINK (X)=P;
LOW BOUND CHECK
        if Q+SIZE (Q)=X then do;
           SIZE (Q)=SIZE (Q)+N;
           LINK (Q)=LINK (X);
                              end
                else do; LINK (Q)=X;
                         SIZE (X)=N;
                     end
end RECOVER;
```

The advantages of this implementation can be said as follows;

i) This method has the storage recovery facility without contrac ion of the whole feature of PASCAL.

ii) It is full use of the storage of run-time stack and heap until the total size is almost exhausted.

iii) It is relatively simple to implement than other storage recovery method applicable to PASCAL.

However, this method has two old problems, dangling references and garbage, because it requires the programmer to keep track of the status of lists, sublists, etc. But old solutions to these problems, reference counts and garbage collection, cannot be applied to PASCAL without contraction of various list processing features. Several difficulties in using and implementing the solutions were discussed in detail by Pratt[11].

7. Conclusion and Remarks

Run-time storage administration needed for recursive nested structure of PASCAL could be accomplished by stack scheme with dynamic link and display management.

The feature of recursive procedure calls was solved by the stack scheme with dynamic link of activation records and the latter gave the advantage of an automatic overlay of data belong to non-activated procedures. And the binding of a name to a location under the most closely nested rule (deep binding) was implemented by the display management using static link and an activation display stored in index registers, which was determined after comparing four available methods on the view of the system efficiency. Finally, the algorithms which could manipulate the dynamic allocation procedure and pointer variable feature upon the previous stack scheme was implemented using heap without loss of the storage efficiency.

References

[1] Aho, A.V., and Ullman, J.D.; Principles of compiler design, Addison-Wesley, Mass., 1977

[2] A.M. Addyman; "A draft proposal for PASCAL," Sigplan. ACM. 15, 4, 1~66, 1980

[3] Bauer, F.L., and Eickel, J.; Compiler construtution: An Advanced Course, Springer Verlag, New York, N.Y., 1974

[4] Bobrow, D. and B. Wegbreit; "A model and stack implementation of multiple environments," Comm. ACM, 16, 10, pp. 591~602, 1973

[5] Britten, D.E., Druseikis, F.C., Griswold, R, E., Hansen, D.R. and Holemes, R.A.; "Procdure referencing environments in SL5," Proc. 3rd ACM. Symposium on Principles of Programming Languages, pp. 185~191, 1976

[6] Dijkstra, E.W.; "ALGOL 60 translation," Supplement ALGOL Bulletin 10, 1960

[7] Gries, D.; Compiler construction for digital computers, Wiley, New York, N.Y. 1971

[8] Gries, D., Paul, M., and Wiehle, H.R.; "Some techniques used in the ALCOR ILLINOIS 70 90," Comm. ACM, 8, 8, 496~500, 1965

[9] Jensen, K., and Wirth, N.; PASCAL User Manual and Report, Springer-Verlag, New York, N.Y., 1975

[10] Knuth, D. [1968] The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading. Mass.

[11] Pratt, T.W.; Programming languages: design and implementation, Prentice-Hall, Englewood Cliffs, N.J., 1975

[12] Randell, b., and Russel, L.J.; ALGOL 60 implementations, Academic Press, New York, 1964

[13] Wirth, N.; "The programming language PASCAL," Acta Informatica 1 : 1, 35~63, 1971a

[14] Wirth, N.; "The design of a PASCAL compiler," Software-Practice and Experience 1 : 4, pp. 309~333, 1971b

[15] W.M. McKeeman; "Symbol table access," In Bauer and Eickel [1974], pp. 253~301, 1974