HEAP STORAGE MANAGEMENT FOR THE PROGRAMMING LANGUAGE PASCAL

by

Dianne Ellen Britton

A Thesis Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 7 5

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Dianne E Britton

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

G. D. Ripley

G. D. RIPLEY
Assistant Professor of
Computer Science

11/21/75
Date

ACKNOWLEDGMENTS

This thesis stems from a project to implement Pascal on the DecSystem-10, using the facilities of the Computer Center at The University of Arizona. David R. Hanson, who conceived of the project, and I worked on it under the general guidance of my thesis director, G. David Ripley.

The numerous technical discussions with Dave Hanson and Frederick C. Druseikis, my husband, were invaluable for the accuracy and completeness of this work. Of course, neither of them is responsible for any remaining inadequacies. I sincerely thank all the above persons and Ralph E. Griswold for their careful reading and perceptive criticism of the first draft of this thesis. Their comments greatly improved the final form.

Finally, I want to express my appreciation to all concerned, especially to Fred, for their encouragement and patience throughout this endeavor.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF ILLUSTRATIONS

ABSTRACT

The programming language Pascal supports pointer variables, whose objects are typically allocated from a heap since their allocation and accessiblility are only indirectly related to block structure. Storage recovery of program-inaccessible heap objects enables a heap to accommodate more objects than otherwise possible. Automatic storage recovery allows maximum recovery without introducing dangling references.

No current Pascal implementations provide for automatic storage recovery, due to the assumption that the provision for automatic storage recovery would delay the implementation. This thesis shows how heap storage management in Pascal can be designed to allow a hierarchy of implementations. The initial implementation, with only allocation operations, is easily and quickly implemented, and provides for later extension to include non-compacting storage recovery by adding to, but not modifying, the initial implementation.

The second implementation includes non-compacting automatic recovery, which requires that accessible objects be distinguishable from inaccessible ones. This

implementation interfaces with Pascal's user-defined data typing facilities. Two approaches to determine accessibility are described: compiler generated type templates processed by an interpreter and compiler generated subroutines.

The final implementation adds compaction to the previous one, but necessitates no modifications to the previous implementation.

CHAPTER 1

INTRODUCTION

This thesis is concerned with heap storage management for programs written in the programming language Pascal. Pascal is probably best known for its user-defined data typing facilities and somewhat less well-known as a language that supports pointer-valued variables. Taking an implementor's point of view, we note that the objects of pointer variables are usually allocated from a storage heap in order to accommodate their unpredictable creation times and life spans. It is precisely the combination of user-defined typing facilities and pointer variables that makes heap storage management in Pascal an interesting topic and, in part, provides the motivation for this thesis.

This work describes three heap storage management systems for Pascal. The first, and simplest, system is a heap storage allocator with the important attribute that its design does not preclude extension to a system that supports automatic storage recovery. The second and third systems are more interesting since they must interface with user-defined typing facilities. The second system is designed by extending the first one with a non-compacting storage

retriever, and the third system adds storage compaction to the second system.

Purpose and Background of This Work

With respect to storage management, Pascal and Algol-68 share similar complexity. Like Pascal, Algol-68 supports user-defined data types (modes), and variables declared "ref" in Algol-68 are roughly equivalent to pointer variables in Pascal. The literature on heap storage management, however, is to be found for Algol-68 (Branquart and Lewi 1970, Goyer 1970, Marshall 1970, Wodon 1970), but not Pascal. It might be argued that since the two programming languages are so similar in this respect, the lack of publications concerning heap storage management for Pascal is due to a reluctance to merely duplicate the Algol-68 literature. The deficiency might better be explained by the fact that heap storage management has not been treated seriously in any current Pascal implementation. The reasoning has been that Pascal is designed for use chiefly as an educational tool and so an easy, fast implementation is most important. The underlying assumption is that an implementation that caters to the special storage management problems associated with user-definable types and pointer-referenced objects can be neither easy nor fast.

We disagree with this assumption. The contribution of this thesis is to illustrate that a heap storage

management system for Pascal that includes automatic storage recovery and compaction is not incompatible with an easy, fast implementation. To this end, the heap storage management algorithms presented in this thesis form a hierarchy of implementations. Initially, one can implement heap storage management with only allocation operations, retaining the possibility of later refinement to include non-compacting storage recovery by adding to, but not modifying, the initial system. In the same manner, a later effort can include compaction. It is shown that the allocation policies of current implementations preclude this sort of hierarchy. Thus we propose a design that admits an initial fast implementation no worse than the current ones, but which is extendable to a more complete heap storage management system as time permits.

The author of this thesis was involved in a project at The University of Arizona to implement Pascal for the DecSystem-10. Although the implementation has not been completed, a considerable amount of insight into the requirements and peculiarities of Pascal, particularly with respect to heap storage management, was gained from the project. In spite of the fact that our understanding of Pascal heap storage management is largely attributable to implementation experience, it must be stressed that the

subject matter here is without regard to any particular Pascal implementation. In fact, the storage management algorithms given here are written in a slight extension of Pascal itself.

Pointer Variables in Pascal

As has been mentioned several times already, Pascal supports pointer variables. The object referenced by a pointer is a variable that, unlike declared variables, is created at runtime asynchronously with respect to block (procedure) entry. In this thesis, the phrase "runtime created variable" always refers to an object referenced by a pointer. Since all objects of pointers are allocated from the heap, the term "heap object" is used interchangeably with "runtime created variable".

A pointer can be declared

```
VAR p: ^t;
```

where p is the variable name and t is the type (either builtin or user-defined) of p's object. Thus, it is determined at compile time that p can only point to objects of type t. Alternatively, a similar effect results from

```
TYPE tpointer = ^t;
VAR  p: tpointer;
```

Here, an explicit user-defined name, tpointer, has been given to the type ^t. A pointer type is just one example of a user-defined type. Other possibilities include record

structures and arrays, which are discussed in a subsequent chapter.

A heap object is created by the builtin procedure NEW. NEW(p) allocates an object of type t from the heap and points p to the new object. Then p^ references the heap as a runtime created variable. For instance, if t is type INTEGER, then the statement p^ := 5 assigns the value 5 to the runtime created variable pointed to by p.

In Pascal, a type is completely specified at compile time. Types are not parameterized in any way, so that, for instance, arrays with dynamic bounds are not possible. This means that the length of any runtime created variable is known at compile time.

Notation and Organization

Throughout the rest of this thesis, it is assumed that the reader is familiar with the Pascal programming language. For a description of the language, see Jensen and Wirth (1974). The algorithms presented are written in an extended Pascal notation, which we have termed "Pascal-X". An explanation of Pascal-X is given in the Appendix.

The goal of the next four chapters (Chapters 2 through 5) is to design a hierarchy of storage management systems. This is achieved through the design of the procedure new, which represents the Pascal heap storage management system.

```
PROCEDURE new(VAR p:^ANY; n:INTEGER);
```

The procedure new is closely related to the Pascal builtin procedure NEW. The parameter p corresponds to the pointer argument to NEW and n is the amount of storage needed to accommodate an object of the pointer. When the compiler encounters a source statement call to NEW, it transforms this statement into a call to new. For instance, if p is type ^t (pointer to a variable of type t), then the Pascal source statement NEW(p) would be transformed into new(p,SIZE(t)).

The procedure new relies on three other procedures to perform storage management. The function allocate, described in Chapter 2, allocates an object from the heap. Chapter 3 describes reclaim, which recovers the heap space occupied by inaccessible objects. The procedure reclaim calls another procedure mark, described in Chapter 4, to determine which objects are accessible. Chapter 5 describes compact, which eliminates external fragmentation resulting from storage reclamation. The last section of each of the Chapters 2, 3 and 5 is devoted to a rather detailed explanation concerning implementation. These sections may be safely ignored by the casual reader without loss of continuity.

Chapter 6 discusses some problems with heap storage management for Pascal, and a summary of this work is presented in Chapter 7.

CHAPTER 2

ALLOCATION

The aim of this chapter is to develop a heap allocation scheme suitable for Pascal. We want our scheme to be easy to implement and to be usable without the additional complications of storage recovery. On the other hand, we desire that the allocation scheme not preclude automatic storage recovery either with or without compaction.

The simplest heap storage management system provides only for allocation. A later section in this chapter discusses the design of the function allocate, which attempts to allocate an object of size n from the heap. A global Boolean variable, SUCCESS, is set to TRUE or FALSE depending on whether or not the allocation is achieved. If an allocation request cannot be satisfied, an error is diagnosed and execution of the user program halts. In the definition of new below, the procedure lackofstorage performs this function.

```
PROCEDURE new(VAR p: ^ANY; n: INTEGER);
BEGIN
 p := allocate(n);
 IF NOT SUCCESS THEN lackofstorage
END;
```

Determining a Suitable Allocation Scheme

Allocation schemes fall into three categories: sequential, linked-list, and bit-mapped. Each of these is examined below for its suitability.

Sequential Allocation

Existing Pascal implementations (for example, Welsh 1972, Wirth 1971) use sequential allocation. The heap is implemented as a contiguous block of storage much like a stack, so that allocation corresponds to a push operation. The advantage of this scheme is the extremely simple and fast allocation algorithm. The disadvantage is that if automatic storage recovery is performed, then compaction must be done at each and every storage recovery. This is because the allocator depends on the contiguity of all free space in the heap: it has no means of coping with blocks of free space (holes) scattered throughout the heap.

Allocation by Freelist

Allocation by a linked-list of free areas (freelist) does not demand that all free space be contiguous, and thus permits storage recovery either with or without compaction. There are a number of variations on freelist allocation. A simple "first-fit" algorithm is described below. (See also Knuth 1973.)

At all times the free areas (holes) of the heap are linked together. Each hole contains its length and the address of the next hole. Initially, the freelist consists of one hole, which is the entire heap. To allocate an area of length n, the freelist is scanned until a hole large enough to accommodate an allocation of length n is reached. The allocation is made from this hole and the freelist is updated to reflect the reduction in the size of the hole.

Storage recovery presents a complication, however. All the space reclaimed by storage recovery must be linked into the existing freelist. This is not difficult, but it may result in an artificial fragmentation of the heap that can prevent later allocation requests from being satisfied: a hole in the heap may be split between two or more entries in the freelist. Thus care must be taken when reconstructing the freelist to insure that the entries in the list accurately reflect the size and number of holes in the heap.

The Buddy System variant of freelist allocation (Knowlton 1965) was designed specifically to solve the hole-consolidating problem. In the Buddy System, allocation is always in blocks of storage whose length is a power of 2. An allocated block of length n begins at an address evenly divisible by $2^{\lceil \log_2 n \rceil}$. A block's "buddy" is the block such that a block and its buddy form an allocatable block of twice the size. A separate freelist is maintained for each

size block so that when a block is freed, it is returned to the appropriate freelist, according to its size. If that freelist, however, contains the block's buddy, then the two blocks are removed from that freelist and the larger block formed by the pair is returned to the appropriate freelist. The process is repeated as often as possible.

The Buddy System, however, is subject to fragmentation that cannot be completely eliminated by compaction:

(1) Internal fragmentation: Since storage is allocated in blocks whose length is a power of 2, when a request is made for an allocation of length n and n is not a power of 2, m addressing units will be wasted, where

$$m = 2^{\lceil \log_2 n \rceil} - n.$$

(2) External fragmentation: Since an allocated block of length n must always begin at an address evenly divisible by $2^{\lceil \log_2 n \rceil}$ (hole consolidation depends on this), compaction cannot be complete.

Bit-Mapped Allocation

In bit-mapped allocation, a map of bits, bitmap, is associated with the heap, which is divided into fixed-size parcels of one or more addressing units. Each bit in the map corresponds to exactly one parcel in the heap, so that the location of a bit in the bitmap determines the address

of the corresponding parcel. A bit in the map is zero if and only if its associated parcel is free.

Allocation consists of simply scanning the bitmap for a contiguous string of zero bits long enough to satisfy the request. The address of the parcel mapped by the first zero bit of the string becomes the address of the allocated storage, and beginning at the first zero bit, the appropriate number of bits are set to one. Note that internal fragmentation occurs only when a parcel size larger than one addressing unit is chosen.

Storage recovery may be effected either by zeroing those strings of bits in the map that represent garbage (inaccessible parcels), or by zeroing the entire map, then setting to one those bits corresponding to accessible parcels. Either way, the strictly linear organization of the bitmap results in immediate consolidation of holes, so that external fragmentation is eliminated.

Thus it appears that the bit-mapped scheme provides easy allocation and permits, without demanding, storage recovery either with or without compaction. Since storage recovery is slightly less complicated than under the free-list scheme, bit-mapped allocation has been chosen for the Pascal heap.

Implementing the Allocator

The declarations of Figure 1 are used throughout the rest of this thesis. The heap is represented by the array heap, whose elements are ADDRESSINGUNITS (e.g., words, bytes), so that the array index corresponds to an address in the heap. The address of the start of the heap is heapstart and the ending address is heapend. Another array, map, represents the bitmap. The size of the bitmap depends on the size of the parcels (parcelsize) and the size of the heap (heapend - heapstart + 1).

```
CONST
 parcelsize      = {addressing units per parcel};

 heapstart       = {starting address of heap};
 heapend         = {ending address of heap };
 heapsize        = {heapend - heapstart + 1};

 mapsize         = {heapsize DIV parcelsize};

TYPE
 bit             = 0..1;
 posint          = 1..MAXINT;

 heapaddr        = heapstart..heapend;

VAR
 { heap:ARRAY[heapaddr] OF ADDRESSINGUNIT; }
 map:   PACKED ARRAY[1..mapsize] OF bit;
```

FIGURE 1: Declarations for Storage Management Algorithms

The utility functions of Figure 2 are assumed to be available. The number of parcels required to accommodate n addressing units of storage is computed by mapwidth(n). This number is the number of bits in the bit map corresponding to a heap object of size n. The procedure setmap(addr,width) sets width bits of the bitmap to one, starting with the bit corresponding to the parcel at heap address addr. In this manner, enough parcels are reserved in the heap to accommodate a runtime created variable of size n, where mapwidth(n) is equal to width.

Figure 3 gives the algorithm that we have chosen for allocation in Pascal. If the algorithm is changed so that

```
FUNCTION mapwidth(size: posint): posint;
BEGIN
 IF size MOD parcelsize = 0
  THEN mapwidth := size DIV parcelsize
  ELSE mapwidth := size DIV parcelsize + 1
END {mapwidth};

PROCEDURE setmap(a: heapaddr; w: posint);
VAR mapindex: 0..mapsize;
    i: posint;
BEGIN
 mapindex := (a - heapstart) DIV parcelsize;
 FOR i := 1 TO w DO map[mapindex + i] := 1
END {setmap};
```

FIGURE 2: Some Utility Functions for Storage Management

```
FUNCTION allocate(n: posint): heapaddr;

{ allocate satisfies an allocation request of n
words from the heap.  It returns the address of
the newly allocated object.

The global variable SUCCESS is TRUE when this
routine returns if and only if the request has
been satisfied. }

LABEL 1;
VAR  i: 1..mapsize;
     j: 0..mapsize,
     holeaddr: heapaddr;

BEGIN
     SUCCESS := FALSE;
     n := mapwidth(n);    { Convert n to the
                            number of parcels. }

     j := 0;              { j is a bit in the map that
                            corresponds to the start of
                            hole. }

     { Perform the search of the bitmap. }
     FOR i := 1 TO mapsize DO
      IF map[i] = 1 THEN j := i
      ELSE IF i-j = n THEN
           BEGIN  { Found a large enough hole }
            holeaddr := heapstart + j*parcelsize;
            setmap(holeaddr);
            allocate := holeaddr;
            SUCCESS := TRUE;
            GOTO 1
           END;
1:
END {allocate};
```

FIGURE 3: The Storage Allocator for the Pascal Heap

the search of the bitmap for a sufficiently large hole begins where the previous search ended, then bit-mapped allocation, in the absence of storage recovery, is comparable to sequential allocation.

CHAPTER 3

STORAGE RECOVERY WITHOUT COMPACTION

This chapter demonstrates how automatic storage recovery without compaction can be achieved for Pascal. In the definition of new, given below, it is shown how automatic storage recovery is integrated into the heap storage management system described in the previous chapter. Recovery of the heap space occupied by inaccessible objects is performed by the procedure reclaim, described in a subsequent section in this chapter. (Of course, even after storage recovery is done, there may not remain enough contiguous free space to satisfy the request. In this case, an error is diagnosed as before.)

```
PROCEDURE new(VAR p: ^ANY; n: INTEGER);
BEGIN
 p := allocate(n);
 IF NOT SUCCESS THEN
  BEGIN
   reclaim;
   p := allocate(n);
   IF NOT SUCCESS THEN lackofstorage
  END
END;
```

In accordance with the implementation hierarchy described in Chapter 1, the restriction is made that the storage recovery algorithm developed here permit subsequent extension to a system that includes compaction.

Why Automatic Storage Recovery?

A heap object is accessible only as long as at least one pointer pointing to it is accessible. A pointer can be a declared variable, in which case it becomes inaccessible as soon as its declaring function or procedure returns, or it can be part of a runtime created variable so that its accessibility is subject to the accessibility of other pointers. Thus, in order to make efficient use of heap storage, some form of storage recovery is required that frees heap storage occupied by only those runtime created variables that are no longer accessible.

Manual Storage Recovery

Since programmers may not understand the behavior of their programs well enough to confidently know when the runtime created variables become inaccessible, explicit deallocation is an inconvenience that many programmers may well decide is not worth the trouble. Nevertheless, those Pascal implementations that have implemented heap storage recovery have provided only explicit means of deallocation.

In Pascal, the standard procedure DISPOSE permits the implementation of manual storage recovery. That is, the programmer indicates by means of DISPOSE that the object of a specified pointer variable is to be considered inaccessible by that pointer. DISPOSE, however, is incompatible with

a sequential allocation scheme, which cannot make use of "holes" in the heap. For this reason DISPOSE has been ignored (semantically) in Pascal implementations.

Instead, Pascal implementations have relied on the non-standard procedure RELEASE, which frees not only the object of a given pointer but also all heap storage allocated after the object. Whenever the programmer can directly specify the deallocation of specific heap objects, there exists the dangling reference problem. That is, an object may be (erroneously) freed while there are still pointers to it. If the freed storage is reused, these remaining pointers no longer reference their intended object. Obviously, RELEASE is very prone to the dangling reference problem and we consider this form of storage recovery unacceptable for heap storage recovery in Pascal.

DISPOSE, which frees only the object specified by the pointer argument, could avoid the problem of dangling references if it used a "use" count. If each heap object had associated with it a count of the number of pointers pointing to it, then DISPOSE could decrement the use count and subsequently free the object only if the use count had dropped to zero. Pascal, however, supports circular list structures. Since a circular list is self-referent, it is possible to have an inaccessible list with non-zero use count (Knuth 1973). Thus, use counts cannot guarantee complete storage retrieval.

Automatic Storage Recovery

Since manual storage recovery schemes are inconvenient for the programmer, and either are dangerous or result in incomplete storage retrieval, we have chosen automatic storage recovery for Pascal. Automatic storage recovery is triggered by an allocation request that cannot be satisfied given the current configuration of the heap; that is, a large enough hole does not exist. Reclamation of storage, without compaction, consists of finding all heap objects that are still accessible to the program and designating the rest of the heap as free. Using the bitmap scheme, the end result of storage recovery is that any bit in the bitmap is one if and only if the corresponding parcel is part of an accessible heap object. Thus, storage recovery in Pascal consists of the following two steps:

(1) All bits in the bitmap are set to zero.

(2) All accessible objects are found, and the corresponding bits of the map are set to one.

Accessibility of heap objects can be defined recursively. A heap object x is accessible if and only if

(1) at least one of the active procedures contains a declared pointer variable whose value is x, or

(2) at least one other heap object y, known to be accessible, contains a pointer whose value is x.

Thus, all accessible heap objects can be located if all pointers in the active procedures and all pointers in any given heap object can be located.

For a heap object, it is to be expected that the compile time specification of the object's type can provide the information required to locate the object's pointers. We show that a similar situation holds for declared pointers in active procedures.

Procedure Data Areas

In Pascal, as in other languages with static scoping and recursive procedures, each activations of a procedure is represented by an entry in a stack of procedure data areas. A procedure data area (PDA) is very similar to a record structured variable in which the parameters and local variables are fields of the structure. For instance, the data area for procedure p declared as

```
PROCEDURE p(x: REAL; y: INTEGER);
VAR z: BOOLEAN;
BEGIN ... END;
```

can be thought of as a variable of a type defined as

```
TYPE pstruct =
      RECORD
       x: REAL;
       y: INTEGER;
       z: BOOLEAN
      END;
```

Thus, the compiler can associate an internally generated record structure type with each procedure declaration in the

same manner that a user-specified type is associated with each variable declaration. The internally generated type is an abstraction of the procedure's PDA and can be used to find any pointers in the PDA of any activation of the procedure.

Parameters passed by reference require special consideration when building the internal type specification for a PDA. To see why this is so, we note the following:

(1) The types of the fields of the PDA type must reflect the actual size of the formal parameters and local variables.

(2) The value stored in the field of a PDA corresponding to a reference parameter is not of the type with which the parameter was declared, but rather is the address of a variable of that type.

For the purpose of building an internal type for a PDA, all reference parameters are considered to be of a type we call REFPARM. This is consistent with the fact that all reference parameters occupy equal amounts of space in their PDA (enough to hold the address of the actual argument), regardless of the declared type of the parameter. Although the actual argument of a reference parameter may be a heap object, so that the value of the reference parameter is an address in the heap, reference parameters are not relevant in determining which heap objects are accessible. This is

because any heap object accessible by virtue of a reference parameter is also accessible by virtue of the corresponding actual argument.

Implementation of Storage Recovery

Using the types for each variable and PDA, it is possible to locate all pointers accessible from a given procedure. In the next chapter, it is discussed in detail exactly how type information can be used for this purpose, but for now we assume that a procedure mark finds all heap objects accessible from a variable and sets the appropriate bits in the bitmap to one. Mark takes two arguments: a variable and a specification of its type. Assuming that mark exists, we now give an algorithm for finding all program-accessible pointers, i.e. all pointers accessible from all active procedures.

First, it is necessary to detail the stack of procedure data areas. For each activation of a procedure, the stack must be used for two purposes.

(1) To provide storage for procedure parameters and local variables. This storage is the PDA and is described by an internally generated type as discussed above.

(2) To retain whatever miscellaneous information is needed by the system, e.g., to maintain the integrity of the stack. The layout of this information is described by the following type:

```
TYPE pdahead =
      RECORD
       pdatype: ^typespec;
       dynamiclink: stackaddr;
       ...
      END;
```

Field pdatype of pdahead indicates the structure of the PDA for the procedure (typespec is defined later). The dynamiclink field points to the PDA of this procedure's caller. At all times during a program's execution, the global variable

```
VAR currentpda: stackaddr;
```

points to the PDA of the current procedure and is used to access the parameters and the local variables of that procedure. The organization of the stack is, reading from left to right, least-recently to most-recently activated procedure:

$$\text{pdahead}_0, \text{pda}_0, \ldots, \text{pdahead}_{\text{CURRENT}}, \overset{\text{currentpda}}{\overset{\wedge}{\text{pda}_{\text{CURRENT}}}}$$

The procedure reclaim, shown in Figure 4 effects automatic storage recovery without compaction. When reclaim returns, the bitmap has been altered so that every bit in the map is one if and only if its corresponding parcel is part of an accessible heap object. In reclaim, mark is relied upon to do the pointer following and bit altering.

```
PROCEDURE reclaim;
VAR thispda: stackaddr;
    thispdahead: ^pdahead;
    i: INTEGER;
BEGIN
 { Zero the bitmap. }
 FOR i := 1 TO mapsize DO map[i] := 0;

 { Locate all active procedure data areas.}
 thispda := currentpda;
 WHILE thispda <> NIL DO
  BEGIN
   thispdahead := thispda - SIZE(pdahead);
   mark(thispda,thispdahead.pdatype);
   thispda := thispdahead.dynamiclink
  END
END {reclaim};
```

FIGURE 4: The Storage Recovery Procedure Reclaim

CHAPTER 4

FINDING THE ACCESSIBLE HEAP OBJECTS

The previous chapter assumes the existence of a procedure called mark, which finds all heap objects accessible from a given pointer and sets the appropriate map bits to one. In order to do this, mark must be able to locate all the pointers in a heap object. In this chapter, we see exactly how type information can be used locate pointers in order to find accessible objects, demonstrating the process with an algorithm for mark.

Structured Types in Pascal

Pascal supports both structured and unstructured types. From our point of view, structured types are the most interesting since only these can contain pointers. Structured types are in general user-defined and there are exactly five kinds of them: sets, pointers, files, arrays, and records. We are not interested in sets since, according to their definition in Pascal, sets of pointers or of other structured types are not possible. Pointer types have already been discussed in previous chapters. Although it is possible to write pointer values onto a Pascal file, pointers stored on a file are never considered when determining

the accessibility of heap objects. Therefore, we ignore the issue of file-structured types.

Pascal arrays are homogenous and fixed-length. Both the element type and bounds specifications are given at compile time. Since any type is valid for an array element, it is possible to define an array of arrays, an array of structures, an array of pointers, etc. An example of an array type declaration is:

```
TYPE vector = ARRAY[1..10] of INTEGER;
```

Record structures in Pascal have non-homogenous components. The number, names, and types of components are fully specified at compile time. Like array elements, record components may be of any unstructured or structured type. Although each component field is bound to only one type, record variants introduce some flexibility into the otherwise rigid typing facilities. In general, a record structure is composed of two portions: a fixed part and a variant part. Two record-structured variables declared with the same type may actually have different components in the variant part. The particular variant in effect for a record-structured variable is determined by the value of a tagfield, a component in the fixed part. The following type statement declares a record type with a variant.

```
TYPE course =
      RECORD
       number: INTEGER;
       CASE credit: BOOLEAN of
        true:  (credits: INTEGER;
                labwork: BOOLEAN);
        false: (hours:   INTEGER)
      END;
```

The fixed portion of type course consists of the field number and the tagfield credit. A value of TRUE in the credit field means that the variant consisting of fields credits and labwork are in effect, whereas a value of FALSE specifies the (other) variant consisting of the single field hours.

In order to locate pointers during storage recovery, it is clear that type information must be available at run-time. This can be accomplished by either of the two methods described in the next two sections. These two methods, the interpretive and compiled methods, are very similar to those described for Algol-68 (Branquart and Lewi 1970, Wodon 1970).

The Interpretive Method

In the interpretive method, the compiler generates a data structure called a "template" for each type. A template provides all the information necessary to locate the pointers in an object of the type and is available to the heap storage management system at runtime. During storage

recovery, an interpreter uses the templates as a guide to find pointers located in heap objects.

Templates

Five sorts of data structuring can be distinguished for the purpose of locating pointers:

(1) Non-NIL pointers usually initiate some special kind of processing and, moreover, their objects may contain pointers.

(2) Arrays may be composed of elements that contain pointers.

(3) Likewise, records may contain pointers in their component fields.

(4) The tagfield value in a record structure determines which variant of the record is currently in effect and must be taken into account.

(5) Other types such as scalars, REFPARMS, etc., never contain pointers.

The following remarks relate to Figure 5 which describes a possible structure for templates.

Templatekind is an enumeration type, where the values for a variable of type templatekind are taken from the following ordered set of literal constants.

(pointer,array,record,tagfield,other)

```
TYPE
 natural = 1..MAXINT;

 field = RECORD
          offset: natural;
          type:   ^template
         END;

 templatekind =
         (pointer,array,record,tagfield,other);

 template = RECORD
             CASE kind: templatekind OF
              pointer:
               (object: ^template;
                objsize: natural);
              array:
               (eltsize: natural;
                noelts:  natural;
                element: ^template);
              record:
               (fields:  ARRAY[1..<nofields>]
                          OF field);
              tagfield:
               (variants:ARRAY[<firsttag>..<lasttag>]
                          OF ^template)
            END;
```

FIGURE 5: Type Specification for a Template

Note that the values of templatekind correspond to the five kinds of data structuring described above.

Just as user-defined types are built on other types, templates are built on other templates. Thus it is natural to expect an interpreter that operates on a template to be recursive.

If a field is part of the fixed portion of a record structure, then we define its offset to be the displacement of the field from the start of the record structure. If a field is part of a variant, then its offset is its displacement from the start of the variant.

A tagfield can only be assigned from a finite set of values, the first of which is firsttag and the last of which is lasttag. The field variants is an array indexed by firsttag through lasttag where each index value selects a different (record-structured) variant.

The Template Interpreter

The procedure reclaim of the previous chapter depends on the procedure mark. In the interpretative method, mark itself is an interpreter whose arguments are an address of an object and the specification of a template. We define the type typespec as follows:

```
TYPE typespec = ^template;
```

As far as reclaim is concerned, each time mark is applied to a PDA, it finds all objects accessible from the PDA and sets the map bits corresponding to the parcels of each object located to one. A more detailed explanation is given below. Mark operates in one of five ways, depending on the kind field of its template argument, as follows:

(1) If the first argument to mark specifies a non-NIL pointer that has not already been processed by mark, then the map bits corresponding to the parcels occupied by the object must be set to one. (The number of parcels can be derived from the objsize field of the template.) Since the pointer's object may itself lead to more pointers, it also must be processed by mark. The address of the pointer's object is the value of the pointer and the template specification is stored in the object field.

(2) If mark is processing an array that contains pointers, then each array element must be processed by mark. The address of the first element corresponds to the address of the array itself. Each subsequent element is at a distance of eltsize from the previous one. The number of elements is available from the noelts field of the template, and the array element template specification is available from element.

(3) When mark processes a record structure (either the fixed part of a record or a single variant), each

component field that contains pointers must be processed by mark. The fields array has one element corresponding to each field in the record structure. The offset of the element specifies the offset of the field from the start of the record structure and the type specifies the template for the field.

(4) Processing a record structure may lead the marking interpreter to process a tagfield. We assume, for illustrative purposes, that all tagfields occupy one addressing unit. We further assume that a variant part lies immediately after the tagfield. The variants field is an array of template specifications for variant part record structures, and the value of the tagfield selects which variant is currently in effect. This variant must be processed by mark. The address of the variant is one addressing unit after the tagfield, and the template for the variant is available from the variants array.

(5) For other data types, mark need not do any processing since there are no pointers to be found.

Figure 6 illustrates the marking interpreter. The interpreter calls itself recursively, using the Pascal system stack to save return addresses. The function marked (not shown) returns TRUE if the object addressed by the argument has already been marked or is in the process of being marked and FALSE otherwise.

```
PROCEDURE mark(v: ^ANY; t: typespec);
VAR i: natural;
BEGIN
 CASE t^.kind OF
  pointer:
   IF v^ <> NIL THEN
    IF NOT marked(v^) THEN
     BEGIN
      setmap(v^,mapwidth(t^.objsize));
      mark(v^,t^.object)
     END;

  array:
   FOR i := 1 TO t^.noelts DO
    BEGIN
     mark(v,t^.element);
     v := v + t^.eltsize
    END;

  record:
   FOR i := 1 TO t^!nofields DO
    mark( v + t^.fields[i].offset, t^.fields[i].type);

  tagfield:
   mark( v + 1, t^.variants[v^]);

  other: { no pointers to find }
 END { CASE }
END { mark };
```

FIGURE 6: The Marking Interpreter

The Compiled Method

In the compiled method, the compiler generates subroutines in the place of templates, where the type information is incorporated into the code of the subroutine. The result is that the compiled method consumes less execution time than the interpretive method, at the expense of space.

In the compiled method, the pdatype specifies a compiler-generated subroutine rather than a template. Thus we define typespec as follows:

```
TYPE typespec = PROCEDURE;
```

where PROCEDURE is a builtin type in Pascal-X. Thus, each pdatype field contains a value specifying the compiler-generated subroutine for the internal PDA type. The procedure mark merely provides an interface between reclaim and the compiled subroutines:

```
PROCEDURE mark(v: ^ANY; t: typespec);
BEGIN t(v^) END;
```

Just as the mark procedure in the interpretive method calls itself recursively, the compiler-generated subroutines call other compiler-generated subroutines (possibly themselves) using the Pascal system stack for temporary storage. Figure 7 gives an example of some user-defined types and the corresponding subroutines. Although the subroutines perform actions comparable to those performed by the marking interpreter, the subroutines are much

```
TYPE
        p = ^q;

        q = RECORD
             a: INTEGER;
             b: p;
             CASE c: BOOLEAN OF
              TRUE:  (d: INTEGER);
              FALSE: (e: p;
                      f: INTEGER)
            END;

        r = ARRAY[0..10] OF q;

PROCEDURE marksubp(v: p);
BEGIN
 IF v <> NIL THEN
  IF NOT marked(v) THEN
   BEGIN
    setmap(v,mapwidth(SIZE(q));
    marksubq(v^)
   END;
END;


PROCEDURE marksubq(v: q);
BEGIN
 marksubp(v.b);
 CASE v.c OF
  TRUE:   ;  { The compiler knows that
             INTEGER variables never
             cause marking. }
  FALSE: marksubp(v.e)
 END { case }
END;

PROCEDURE marksubr(v: r);
VAR i: INTEGER;
BEGIN
 FOR i := 0 TO 10 DO marksubq(v[i]);
END;
```

FIGURE 7: Some Compiler-Generated Marking Subroutines

simpler. Each subroutine takes only a single argument and is tailored to that argument's type. Since each subroutine knows *a priori* the type of its argument, this information is implicitly carried in the code of the subroutine itself, whereas in the interpretive method, the type information has to be tediously extracted from a template.

CHAPTER 5

COMPACTION

Unless compaction is performed, storage recovery is likely to result in external fragmentation. In this situation, where free space occurs in many small blocks of parcels scattered throughout the heap, it may be impossible to satisfy an allocation request for a given number of contiguous parcels even though the total available space is sufficient to satisfy the request. External fragmentation is eliminated by means of compaction: all objects in use are rearranged so that they occupy contiguous parcels in the heap. Thus, after compaction, all used parcels occur as one contiguous region at one end of the heap while all free parcels occur at the other.

In this chapter, the storage management system of the previous chapters is extended to include a procedure called compact. The definition of new, given below, demonstrates the final component in our hierarchy of heap storage management systems for Pascal. Since compaction is an expensive process, it is done only when storage recovery without compaction does not yield enough contiguous free space to satisfy the allocation request. But compaction,

too, may fail to yield the required space, in which case the error is diagnosed as before.

```
PROCEDURE new(VAR p: ^ANY; n: INTEGER);
BEGIN
 p := allocate(n);
 IF NOT SUCCESS THEN
  BEGIN
   reclaim;
   p := allocate(n);
   IF NOT SUCCESS THEN
    BEGIN
     compact;
     p := allocate(n);
     IF NOT SUCCESS THEN lackofstorage
    END
  END
END;
```

Choosing a Compaction Method

Compaction is composed of three steps: calculating new addresses for the accessible objects, moving the accessible objects to their new locations, and updating pointers to reflect the new addresses of their objects. There are two general classes of compaction algorithms, depending on the order in which object moving and pointer updating is done. One class of algorithms is exemplified by the SNOBOL4 implementations (Dewar 1971, Griswold 1972, Hanson 1975), where the pointers are updated before the accessible objects are actually moved. The Haddon-Waite algorithm (1967), which has been suggested for use in ALGOL-68 implementation (Branquart and Lewi 1970, Wodon 1970), moves the accessible objects and then updates the pointer values. Both classes

of algorithms permit correct processing of arbitrary list structures, such as those found in Pascal programs.

SNOBOL4-Type Compaction

SNOBOL4-type compaction (S-compaction) actually has two variants. The method used by the macro implementation (Griswold 1972) is perhaps more straightforward, so it is explained first. After the marking phase, the heap is scanned in a linear fashion for all marked, i.e., accessible, objects. For each such object, a new address is calculated and stored within the object itself. In the macro implementation, pointers are self-identifying, so that it is possible to find all pointers with a linear scan of the heap. Thus, at the completion of the new-address computation, pointer updating is done by means of another linear sweep. Each pointer in each accessible object is updated to the new-address value stored in the pointer's object. A final linear scan moves all marked objects to their new addresses.

Although neither SPITBOL (Dewar 1971) nor SITBOL (Hanson 1975) have self-identifying pointers, they nevertheless achieve pointer adjustment with a simple linear scan. In this case, the marking phase links together all pointers to a given object and stores the head of the list in the object. Pointer updating is done by linearly scanning the heap for all accessible objects, calculating a new address

for each such object and updating all pointers on the list originating in the object with the new address. In this variant, the new address is not stored with the object, but is recomputed with the final (compacting) linear scan.

Haddon-Waite Compaction

After the marking phase, Haddon-Waite compaction (HW-compaction) does a linear scan through the heap, during which each contiguous group of used parcels (one or more marked objects) is moved as far as possible toward one end of the heap. For each such move, the old starting address of the group of used parcels is stored in a break table, together with a correction number equal to the number of addressing units the group was moved. Then all pointers are found in a manner analogous to the marking phase. As each pointer is found, its value (or next lowest one) is found in the break table and the pointer value is updated by the corresponding correction number.

Our Choice for Pascal

The hierarchy of Pascal heap storage management systems described in this thesis constrains the choice of compaction algorithms. The addition of compaction must not require any modifications of the existing storage allocation and recovery algorithms. It is shown below that HW-compaction meets this criterion, whereas S-compaction does not.

New-address calculation in HW-compaction is immediately compatible with our bitmap scheme. Addresses of blocks of contiguous used parcels, not of individual marked objects, are required during the linear scan and these addresses are available directly from the bitmap. On the other hand, S-compaction requires that individual marked objects be found during the linear scan to compute new addresses. The addresses of marked objects are not immediately available from the bitmap. This means that in order to implement S-compaction the length of each object must be deducible from the object itself, requiring the addition of an overhead field in each heap object.

During pointer adjustment, HW-compaction derives new address information from the break table. It has been shown (Haddon and Waite 1967) that if a parcel is large enough to hold one break table entry (two addresses), then the table can be built in the free space plus a small amount (one or two table entries) of temporary storage. In S-compaction, however, an additional field is required in each object to accomplish pointer adjustment. This field contains either the new address of the object or heads the list of all pointers to the object. Thus, the use of S-compaction in our storage management system would necessitate a change to the layout of heap objects.

The disadvantage of HW-compaction lies entirely in execution speed. Locating pointers during the pointer adjustment phase requires that a process similar to marking be done. That is, a template interpreter or group of compiler-generated subroutines must be used to find the pointers. This process is certainly slower than the linear scan used in S-compaction for pointer updating. Pointer adjustment is further slowed by the table lookup required, but a reasonable lookup algorithm, like a binary search on a sorted break table, can minimize the additional execution time.

However, the fact that the implementation of HW-compaction requires absolutely no change to the storage allocation or recovery algorithms developed in previous chapters far outweighs the disadvantage of somewhat slower execution time. After all, compaction is to be done only when absolutely necessary, that is, when the storage recovery procedure reclaim has failed to yield enough space to satisfy an allocation request.

It should be noted that although our implementation of HW-compaction uses the Pascal system stack during the pointer adjustment phase, the system stack will not overflow during compaction; if the marking phase has been completed without overflowing the stack, then pointer adjustment during compaction will not result in stack overflow either.

Implementation of Compaction

In our implementation of compaction, the break table building and accessing is done by addbreak, sortbreaks and corrected. The addbreak procedure builds the break table in the otherwise unused heap holes by adding table entries one at a time. The algorithm is described by Haddon and Waite (1967) and is not restated here. Sortbreaks orders the break table and can be implemented with any sorting algorithm that requires only a fixed amount of storage in addition to the table itself. Likewise, the corrected function can be implemented in a number of ways, since it is essentially a search algorithm. Corrected finds the break table entry for a given address and returns the updated address computed from the correction displacement.

Figure 8 gives the procedure compact, which implements HW-compaction for our storage configuration. The procedure moveparcel moves the parcel at the heap address corresponding to the first argument to the heap address specified by the second argument.

The update procedure is similar in spirit to mark, but its purpose is to update pointer values. Like mark, update spends most of its time following pointers and can be implemented with either the interpretive or compiled method. If the interpretive method is used, then update is just a different interpreter that operates on the same templates

```
PROCEDURE compact;
VAR oldposition, newposition, i: 1..mapsize;
BEGIN { compact }
 newposition := 1;

 FOR oldposition := 1 TO mapsize DO
   IF map[oldposition] = 1 THEN
    BEGIN

     { If a break point has been reached,
     make a table entry }
     IF oldposition = 1
      THEN addbreak(oldposition,newposition)
     ELSE IF map[oldposition - 1] = 0
      THEN addbreak(oldposition,newposition);

     { Compact this portion of storage }
     moveparcel(oldposition,newposition);
     newposition := newposition + 1
    END;

 { Sort the break table, for efficient searching }
 sortbreaks;

 { Zero the bitmap }
 FOR i := 1 to mapsize DO map[i] := 0;

 { Update pointers }
 thispda := currentpda;
 WHILE thispda <> NIL DO
   BEGIN
     thispdahead := thispda - SIZE(pdahead);
     update(thispda,thispdahead.pdatype);
     thispda := thispdahead.dynamiclink
   END
END;
```

FIGURE 8: The Compact Procedure

used by mark. As a matter of fact, update is exactly like mark, except for pointer variable processing. Figure 9 illustrates the action taken for pointer variables.

With the compiled method, the same sort of situation holds for update as for mark: update itself degenerates into an interface between compact and the generated subroutines, which are extremely simple. Since the pdatype field of the PDA header references a marking routine, this field cannot be used as the second argument to update. Thus another field for type pdahead must be defined when using compiler generated subroutines for pointer updating. Figure 10 is an example of an updating subroutine generated by the compiler in the compiled method.

```
PROCEDURE update(v: ^ANY;
                 t: ^template);

VAR i: natural;

BEGIN
 CASE t^.kind OF
   pointer:
        { v^ is a pointer }

        IF v^ <> NIL THEN
           {look up v^ in table, add correction factor}
           v^ := corrected(v^);
           IF NOT marked(v^) THEN
                BEGIN
                 setmap(v^,mapwidth(t^.objsize));
                 update(v^,t^.object)
                END;

   array:  ...
   record: ...
   tagfield: ...
   other: ...
 END { CASE }
END { update };
```

FIGURE 9: The Update Procedure (Interpretive Method)

```
TYPE
 p = ^q;

 q = RECORD
      a: INTEGER;
      b: p;
      CASE c: BOOLEAN OF
       TRUE:  (d: INTEGER);
       FALSE: (e: p;
               f: INTEGER)
     END;

{ upsubp updates the pointers in an object
 of type p. }
PROCEDURE upsubp(v: p);
BEGIN
 IF v <> NIL THEN
  BEGIN
   v := correction(v);
   IF NOT marked(v) THEN
    BEGIN
     setmap(v^,SIZE(q));
     upsubq(v^)
    END
  END
END;
```

FIGURE 10: A Compiler-Generated Updating Subroutine

CHAPTER 6

DISCUSSION

The previous chapters have ignored some important problems associated with heap storage management. This chapter makes note of these problems and offers some suggestions for their solution.

Incorrect Reference to Fields of Variants

In Pascal, there is a substantial amount of information available at compile time. In particular, variables and pointers are strictly bound to their types at compile time. We have seen that this binding can be used to advantage for finding pointers and accessible heap objects during storage recovery. Most Pascal implementations, however, are very careless about ensuring that all accesses to fields in a variant are valid. For instance, the following "incorrect" program segment will pass without notice through most Pascal systems.

```
TYPE  struct = (square,cube);

      object = RECORD
                 CASE form: struct OF
                   square: (area: INTEGER);
                   cube: (volume: INTEGER)
               END;
```

```
VAR x: object;
  ...
x.form := square;
x.volume := 25;
```

The reference to x.volume is inappropriate since x.form is square. n general, incorrctfield references of this sort cannot be prevented by compile-time checking. The danger for automatic storage recovery is that a pointer-typed field can be given a non-pointer value, which is disastrous for the marking algorithms.

The lack of control exercised by Pascal implementations over field accessing in variants seems to have been encouraged by the unrestricted availability of the tagfield to the Pascal programmer. The programmer is responsible for setting tagfield values and is expected to maintain valid references to the fields of variants. If this attitude is maintained in conjunction with an automatic storage recovery system, storage recovery can be guaranteed correct only for those programs that correctly address fields of variants. The solution is for the compiler to generate a tagfield value check along with each variant field access.

Initialization of New Objects

If automatic storage recovery is to work at all, some initialization must be done when an object is allocated. Pointers within the object must be set to NIL, since

the process of finding accessible objects during storage recovery requires that a pointer have a non-NIL value if and only if it points to an object in the heap. Since variable initialization is not a property of Pascal, it must be specially included if automatic storage recovery is implemented.

Temporary Pointer Variables

In the storage recovery scheme presented in this thesis, all accessible objects are found by a search that originates with pointer variables in the stack of procedure areas. All such pointers have been assumed to be explicitly declared in the Pascal program. We have yet to discuss the existence of pointer-valued temporary variables, which also lead to accessible heap objects.

Although evaluation of arithmetic expressions cannot produce any intermediate (temporary) pointer-valued results, function and procedure argument evaluation can. For instance, in the statement

```
x := f(g(y),h(z))
```

temporary variables corresponding to g(y) and h(z) are needed as arguments to f. Consider what happens if g is pointer-valued and if the evaluation of h(z) initiates an automatic storage recovery: heap objects that are accessible from the value g(y) must be found during the marking process.

The problem can be easily solved. We need only consider temporary variables with the following two properties: they are arguments to a procedure or function and they themselves are the result of a function. Each such temporary variable is given an implicit declaration at compile time, thereby increasing the size of the procedure data area and adding an additional field to the generated record structure type for the PDA. For example, the procedure declaration

```
PROCEDURE P(x: REAL);
VAR y,z: INTEGER;
BEGIN
 ...
 x := f(g(y),h(z));   { f,g, and h are
                        INTEGER functions }
 ...
END;
```

would result in the following generated type for the procedure's PDA:

```
TYPE Pstruct =
       RECORD
        x: REAL;
        y,z: INTEGER;
        t1: INTEGER;   { result of g(y) }
        t2: INTEGER;   { result of h(z) }
       END;
```

The locale within a program over which a temporary variable is known is relatively small. In particular, the storage recovery process should not consider objects of "out of date" temporary pointers to be still accessible. Thus, when a temporary pointer variable is no longer of use, the compiler should ensure that it is set to NIL.

External Functions and Procedures

The integrity of the storage recovery mechanism designed in this thesis relies heavily on the compile-time type checking facilities of Pascal. The addition to the language of external procedures, i.e., separately compiled procedures, limits the effectiveness of compile-time type checking. The danger for storage management is that an external function or procedure call can result in a non-pointer value being assigned to a pointer. For example, consider the following program segment,

```
PROGRAM p(OUTPUT);
        ...
FUNCTION g(VAR z: ^something): ^something;
 EXTERNAL;

VAR x,y: ^something;
        ...
 x := g(y)
```

where g is a separately compiled external function, declared

```
EXTERNAL FUNCTION g(VAR z: REAL): INTEGER;
BEGIN
  z := 53.8;
  g := z + 45.2
END;
```

Since g is external, compile-time type checking for the statement x := g(y) in program p is useless. There is no way to ensure that g really returns ^something or that it expects its argument z to be ^something, except by runtime type checking.

The Root of the Problems

This chapter shows that pointer variables in Pascal are not backed, in terms of language design, by a commitment to storage recovery. This is evident because variable initialization and tagfield value checking are clearly necessary for storage recovery but are not part of the Pascal language design. None of the problems presented are a result of the particular storage management scheme used in this thesis, but are inherent to any automatic storage recovery scheme in Pascal. Fortunately, there exist *ad hoc* solutions, as we have indicated.

In fact, problems are presented even for any *explicit* (manual) storage recovery mechanism. Explicit deallocation with use counts, as described in Chapter 3, implies some initialization of newly allocated heap objects. Tagfield value checking is needed to insure that the argument to the builtin DISPOSE function really is a pointer. In general, the problems can be traced to the fact that storage recovery has been poorly integrated into Pascal, regardless of the inclusion of the builtin heap deallocation procedure DISPOSE.

It should be noted that external functions and procedures are poorly integrated into Pascal with respect to the data typing facilities of the language. It is only as a consequence of this that external routines present problems for storage recovery.

CHAPTER 7

SUMMARY

In this thesis we have designed a heap storage management system for Pascal that includes allocation, storage recovery and compaction. Due to Pascal's similarity to Algol-68 with respect to pointer-variable and data-typing facilities, the heap storage management algorithms described here for Pascal are similar to those described by Branquart and Lewi (1970) and Wodon (1970) for Algol-68. The most important aspect of the algorithms presented here, however, is that they have been chosen so as to permit implementation of heap storage management with successive levels of refinement.

Thus, our design consists of an initial implementation which supports only allocation, but which can later be refined to include storage recovery. Likewise, the storage recovery algorithms permit the implementation of compaction to be deferred. Moreover, the compaction algorithm given here does not require that either the allocation or storage recovery algorithms be changed.

Our heap in Pascal is configured with a bitmap. This decision was made after evaluating sequential,

freelist, and bit-mapped allocation schemes with respect to our goals. Sequential allocation was rejected because, although allocation is simple and fast, compaction is demanded at every storage recovery. Freelist schemes were rejected because storage recovery results in an artificial fragmentation of storage unless special care is taken to avoid the problem. With a bit-mapped scheme, however, allocation is comparable to using a freelist but artificial fragmentation does not arise. Moreover, storage recovery both with and without compaction is possible.

Since explicit deallocation of heap objects by the programmer permits dangling references, and since storage recovery through the maintenance of use counts in heap objects cannot always be complete in the case of circular list structures, an automatic storage recovery scheme was chosen for Pascal. Storage recovery is triggered by an unsatified allocation request. The problem of marking accessible heap objects is simplified by the compile time binding of user-defined types in Pascal. A marking interpreter, which operates on compiler-generated type templates, is at the heart of the storage recovery scheme. It is possible to substitute compiler-generated subroutines for the marking interpreter.

The compaction algorithm chosen is essentially the one developed by Haddon and Waite (1967). It was chosen

because it can be imbedded in the rest of the heap storage management system without necessitating changes to the existing algorithms. The fact that this compaction method is slower than others is not a serious drawback since the storage management system has been designed so that compaction is performed only when storage recovery fails to yield enough space to satisfy an allocation request.

It is hoped that this thesis will encourage the development of Pascal implementations that will eventually include not only heap allocation but also heap storage recovery and compaction. For this reason, we offer a design that permits a usable heap storage management system long before the system is complete.

APPENDIX

DESCRIPTION OF PASCAL-X

Many high-level languages are incapable of describing themselves. Even languages that permit self-compilation may lack facilities for self-description of the runtime system. Pascal is an elegant language for describing many algorithms, but its facilities are inadequate in some instances for describing heap storage management algorithms. For instance, we wished to write an algorithm that operates on a variable, regardless of its type, but the compile-time type checking facilities forbid this. We needed procedure-valued variables, but Pascal only allows procedure-valued formal parameters (not variables in general).

Consequently, we extended the Pascal syntax to accommodate our needs, and called the resulting notation Pascal-X. We emphasize that Pascal-X should be regarded only as a notational convenience: we are not proposing that the heap storage management system for Pascal be implemented in Pascal or even in Pascal-X (if it existed). In fact, we have given little thought to the problem of implementing Pascal-X and consider it to be beyond the scope of this work.

How Pascal-X Differs From Pascal

Most of the Pascal-X extensions are concerned with circumventing the rigidity of compile-time type binding.

TYPE Statement

The Pascal-X TYPE statement allows incompletely specified types through the use of parameters in a type specification. If we consider a variable v whose type is an instance of a parameterized type t, then a value for a parameter p of type t is bound to the variable at the time of the variable's allocation, and is available to the programmer as v!p. In order to specify that an identifier occuring in a type declaration is a parameter to the type, the parameter identifier is enclosed in angular brackets. An angular bracket construction is again used when a variable of a parameterized type is allocated. For example,

```
TYPE varying =
        ARRAY[1..<size>] OF INTEGER;
```

The type varying includes the parameter size. Consider the following object:

```
VAR x: varying<size=50>;
```

Here x is an fifty-element array of integers, and x!size has value 50. Likewise the following Pascal-X statements

```
VAR n: INTEGER;
    p: ^varying;
   ...

NEW(p<size=n>);
```

result in p pointing to an integer array of n elements, and p^!size has the value of n at the time of allocation of p^.

^ANY Builtin Type

The builtin type ^ANY specifies a pointer to any Pascal type. Suppose y has been declared as follows:

```
VAR y: ^ANY;
```

Then y^ is assumed by the compiler to be of the type required by the expression in which it appears. For instance, if procedures p and q are declared

```
PROCEDURE p(a: ^usertype);
   ...
PROCEDURE q(b: INTEGER);
```

then p(y^) and q(y^) are both acceptable procedure calls.

Syntactically, the type ^ANY is to be considered a single entity. That is, although variables may be pointers to any type, ANY does not specify the type of a variable. Consequently, the following Pascal-X statements are incorrect and will be flagged by the compiler. Assume that y is declared as in the above example.

```
VAR     z: ANY;     { There is no builtin type ANY }
NEW(y);             { What type would y^ be if this
                      statement were valid? }
```

Address Arithmetic

Address arithmetic is included in Pascal-X and is invoked when at least one argument of an arithmetic expression is a pointer. In the following example, assume a machine with an addressing width of 1. Then, given the declarations

```
TYPE t = ARRAY[0..10] OF INTEGER;

VAR a: ^t;
    b: ^INTEGER;
```

and statements,

```
NEW(a);
b := a + 3;
```

we have b^ equivalent to a^[3].

SIZE Builtin Construct

The function-like construct SIZE(type) gives the maximum size (in addressing units) for a variable of the specified type.

INTERNALPROC Builtin Type

The set of builtin types is extended in Pascal-X to include the type INTERNALPROC. An INTERNALPROC variable is a procedure that is spontaneously generated by the compiler (see Chapter 4) and that has no free variables. Since there are no free variables, the procedure can be invoked regardless of the state of the current environment.

REFERENCES

BRANQUART, P. AND J. LEWI (1970) "A Scheme of Storage Allocation and Garbage Collection for Algol-68" in Algol-68 Implementation, North-Holland Publishing Company, Amsterdam.

DEWAR, ROBERT B. K. (1971) "SPITBOL -- Version 2.0", SNOBOL4 Project Document S4D34, Illinois Institute of Technology, Chicago, Illinois.

GOYER, PIERRE (1970) "A Garbage Collector to be Implemented on a CDC 3100" in Algol-68 Implementation, North-Holland Publishing Company, Amsterdam.

GRISWOLD, RALPH E. (1972) The Macro Implementation of SNOBOL4, W. H. Freeman, San Francisco.

HADDON, B. K. AND W. M. WAITE (1967) "A Compaction Procedure for Variable-Length Storage Elements", Computer Journal, Vol. 10, No. 10, pp. 162-165.

HANSON, DAVID R. (1975) "Dynamic Allocation and Reclamation of Variable-Size Storage Elements in SITBOL", SNOBOL4 Project Document S4D52a, The University of Arizona, Tucson, Arizona.

JENSEN, KATHLEEN AND N. WIRTH (1974) Pascal User Manual and Report, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg.

KNOWLTON, K. (1965) "A Fast Storage Allocator", Communications of the ACM, Vol. 8, No. 10, pp. 623-625.

KNUTH, D. E. (1973) The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Edition, Addison-Wesley, Reading, Massachusetts.

MARSHALL, S. (1970) "An Algol-68 Garbage Collector" in Algol 68 Implementation, North-Holland Publishing Company, Amsterdam.

WELSH, J. (1972) "A Pascal Compiler for the ICL 19000 Series Computer", Software-Practice and Experience, Vol. 2, No. 1, pp. 73-77.

WIRTH, N. (1971) "The Design of a Pascal Compiler", Software-Practice and Experience, Vol. 1, No. 4, pp. 309-333.

WODON, P. L. (1970) "Methods of Garbage Collection for Algol-68" in Algol-68 Implementation, North-Holland Publishing Company, Amsterdam.