ERROR CHECKING WITH POINTER VARIABLES

Marvin V. Zelkowitz, Paul R. McMullin, Keith R. Merkel, Howard J. Larsen

Department of Computer Science and Computer Science Center
University of Maryland, College Park, Maryland 20742

The use of pointer variables in a programming language often results in a difficult class of errors to detect. Pointers may point to storage that no longer is allocated, or storage may be allocated as one data type and accessed as another. This report describes the implementation of pointers in the PLUM PL/1 compiler such that all error conditions are detected. In addition, preliminary tests with PLUM seem to indicate that the total checking of pointers is not as expensive an operation as it first appears.

1. Introduction

The ability to process an arbitrary number of instances of a complex data structure frequently requires the use of pointer variables in a higher level programming language. In such cases an arbitrary data structure is allocated, and a pointer variable is set to point to that structure. The fields within this structure can then be accessed via the pointer's value.

The use of pointer variables frequently leads to several types of errors. For example, in a block structured language like PL/1, storage is allocated on block entry and deleted on block exit. If a pointer variable of an outer block is set to point to a variable in an inner block, then on block exit the pointer will still have a value even though the storage for the block has been deleted.

In general there are two techniques for storage allocation: retention and deletion [1]. The technique described in the preceding paragraph is an example of the deletion strategy, with the pointer resulting in a dangling reference - since it has a value that points to unallocated storage. In the retention strategy, storage remains allocated until the last pointer to it disappears. The contour model [3] is an example of the retention policy.

While the retention policy avoids the dangling reference problem, the deletion policy has certain advantages. For example a stack can be used for block structured storage allocation rather than a general heap storage which is necessary for retention. However, checking for invalid references is more difficult with the deletion policy.

Over the past six months pointer variables have been added to the PLUM PL/1 compiler for the Univac 1100 series computer at the University of Maryland. Since PLUM's runtime environment is based upon a stack implementation, the deletion policy was chosen as the basic mechanism. Also since PLUM is a diagnostic system, total error checking in pointer variable usage was desired. This report is a description of the algorithms implemented, and a preliminary report on the results of that implementation.

2. PL/1 Pointers

In PL/1 local storage is stack oriented (as in ALGOL) while pointers are used to allocate from a heap. Pointers contain the location of the storage that is allocated, while a based variable is used to describe that storage. For example, consider the statements:

```
DECLARE X BASED CHARACTER(8);
DECLARE P POINTER;
ALLOCATE X SET(P);
```

The first declaration declares X to be a template describing a character string of length 8. Notice that it is simply a description of storage - it is not storage itself. The second declaration declares P to be a pointer variable whose own storage is allocated when the block containing P is entered. The third statement allocates an area described by X (character string of length 8) and assigns its address to P.

The storage allocated to P can be accessed via a based reference as in:

```
DECLARE X BASED CHARACTER(8);
DECLARE Y BASED CHARACTER(8);
DECLARE P POINTER;
. . . (P->X) . . .
```

In this example, (P->X) is the X component of the storage pointed to by P. Since X and Y have exactly the same description, in this case (P->X) could have been replaced by (P->Y).

This example points out an important feature of PL/1 pointer variables - pointers are not typed. That is every usage of a pointer variable must be followed by a based variable describing the contents of the structure pointed to.

In considering all that has been written so far, the following errors may exist in using PL/1 pointers:

1. Mixed Mode - The storage pointed to by a pointer is of a different data type than the based variable used to access it.

2. Dangling References - A pointer points to storage that was freed, either implicitly by exiting the block containing the storage or explicitly via a FREE statement.

3. Inaccessible References - Storage may be explicitly allocated via an ALLOCATE statement, and then the pointer is set to some other value. In this case the storage is forever lost since it can never be returned to the system.

The first class of errors above is generally particular to PL/1 since many languages force pointers to be type checked at compile time [6]. However, the latter two classes are not particular to PL/1, and can occur in almost any language that has pointers.

3. Pointer Error Checking

3.1. Mixed Mode

In order to eliminate this form of error, typed pointers were first proposed. Declarations like the following were first considered:

```
        DECLARE P POINTER (CHARACTER(8));
```

This would declare P to be a pointer which could only point to a character string of length 8. However, this is not PL/1. A similar effect, however, could have been achieved via the PL/1 declaration:

```
        DECLARE X BASED(P) CHARACTER(8);
```

In this case X is a based (template) character string of length 8, and is pointed to by P. If P may not point to any other data type (e.g. DECLARE Y BASED(P) REAL would be illegal), then it is possible to type check all references at compile time. All that would be necessary at runtime would be a check for valid pointer references.

If we had been designing a new language, this approach would have been taken; however, the goal of PLUM is to be a diagnostic PL/1 checkout compiler. Because of this, and since runtime execution speed was not of primary importance, it was decided that compile time type checking was too restrictive, and we opted for a runtime check.

In order to provide this runtime check, pointers were implemented with the following runtime storage structure:

```
        +----------+----------+
        ! SYMBOL   ! STORAGE  !
        ! TABLE    ! ADDRESS  !
        +----------+----------+
        ! LINKAGE             !
        !         INFORMATION !
        +----------+----------+
```

When storage is allocated, besides setting the address of the storage into the STORAGE ADDRESS field of the pointer variable, the address of the based variable's symbol table entry is also included. (Since PLUM is a diagnostic system, a runtime symbol table exists for all variables in the program.) Every use of a based reference is preceded by the code:

```
 CALL RESOLVE(pointer, based reference)
```

If the symbol table address of the based variable used with the pointer reference is the same as that contained in the SYMBOL TABLE field of the pointer variable, then a valid reference exists, and the operation proceeds normally. If the references do not match, then a runtime routine compares the attributes of both based variables and checks for compatibility.

For example, consider the example:

```
      DECLARE X BASED FIXED BINARY;
      DECLARE Y BASED FIXED BINARY;
      DECLARE P POINTER;
      . . .
      ALLOCATE X SET(P);
      P->Y = P->X + 2;
```

In this example, the ALLOCATE statement sets up P with the value:

```
      +----------+----------+
      !    X     ! address  !
      !          ! allocated!
      +----------+----------+
```

Before each usage of a based reference, the RESOLVE routine is called. In the assignment statement, the code generated is:

```
  CALL RESOLVE(P,X)
  LOAD register,P->X
  ADD register,2
            /* compute RHS */
  CALL RESOLVE(P,Y)
  STORE register,P->Y
            /* set assigned value */
```

In this case, the second call to RESOLVE (e.g. RESOLVE(P,Y)) takes significantly more time than the first call to RESOLVE. Since P points to a variable of format X, the runtime routine must check that X and Y have exactly the same form. For PL/1 structures, this results in a tree walking algorithm that checks corresponding leaf nodes of the structure for compatibility.

Note that this implementation is a restriction of PL/1. PLUM does not allow a user to set up a structure and then access it via a different data type (e.g. setting up a based character string, then accessing the individual bits of the string by overlaying a bit string on top of it). If the user wishes to perform that function, then the UNSPEC builtin function can be used. By using UNSPEC, the user is explicitly declaring that type checking is to be ignored - PLUM will not allow the user (implicitly or inadvertently) to ignore type checking.

The RESOLVE routine was relatively easy to implement in PLUM since a runtime symbol table exists; however, it is also possible to implement in a standard compiler. If a based variable is a simple scalar variable, then the field SYMBOL TABLE could simply be an encoding of the attributes of the variable. If the based variable is a complex structure, then SYMBOL TABLE could point to a condensed list of attributes for each element of the structure.

3.2. Dangling References

The most difficult aspect of pointers is how to decide whether they point to valid addresses. In Algol-68 [4] this is accomplished by not allowing pointers to point to variables declared in an inner block. In another scheme [5] Lomet uses tombstones to access storage by accessing the allocated storage indirectly via these tombstones. However, the tombstones must remain even if the storage for the variable is deallocated. In [2], a brief algorithm is presented for solving this problem in EULER; however, the details of such an implementation are lacking.

In PL/1 storage is allocated in areas. Thus a user can specify an area variable, which is initially an unformatted hunk of memory, and add the phrase:

```
        IN(area variable)
```

to the ALLOCATE statement. In order to accommodate these, all storage allocation in PLUM is within areas. Three different areas exist.

The first type of area is the system area. This is all of memory outside of the user's stack. All storage that is allocated without an explicit area specified comes from this system area.

A second type of area is the area variable - a block of storage set aside by the programmer.

A third form of area is the stack area associated with a given block. Since the ADDR builtin function allows for accessing the address of any local variable, this stack area is also considered to be an area variable. As will shortly be explained, when any area variable is deallocated, all pointers pointing into it are found. By making normal block termination the same as area deallocation, the same algorithms can be used.

In each area variable (and stack activation record), all allocated storage is chained together so it is possible to find all storage allocated within a given area. In addition all pointers that point to a given variable are also linked together via the LINKAGE INFORMATION field of the pointer. Thus in:

```
   ALLOCATE X SET(P);
   Q=P;   /* Q is a pointer also */
```

the storage for X (figure 1) will contain a pointer to Q, and Q will have a pointer to P. When the storage for X is freed, either by a FREE statement, or by exiting the block containing the address pointed to, this list is scanned, and all pointers on it are set to a null value. (Actually they are set to a value that will invoke an error message the next time that they are examined by the resolve routine.)

3.3. Inaccessible Storage

The previous section also shows how inaccessible storage is discovered. Whenever a pointer's value is altered it is deleted from the list of pointers pointing to a given based allocation. If this list now contains no entries, then nothing points to the allocation any longer, and is therefore inaccessible.

In addition to the linkage field in pointers, all pointers declared within a given block (or area) are linked together. Whenever a block is exited (or area deallocated) this list is scanned, and any pointers still containing non-null values are unlinked from their based references. This enables the discovery of inaccessible references caused when the last pointer to it is deallocated.

It should be noted that this process is recursive. It may turn out that when a pointer is deallocated, a based area may now be inaccessible. It is then necessary to go through the area and set to null any pointer pointing into it. In addition any pointers in the area are unlinked from what they point to, which may make some other area inaccessible, etc. (See figure 2.)

There is still one type of error that PLUM doesn't catch. If a ring structure is allocated, and then the last reference to any element of the ring is deallocated, then each allocation within the ring will have a pointer to it, but nothing will point to the ring itself. Periodic garbage collection could work in this case since all allocated storage is chained together; however, this has not as yet been done. Unlike dangling reference errors, this error will not permit invalid answers since no illegal references are allowed - only the loss of some storage.

For both dangling references and inaccessible storage, code was added to the RESOLVE routine described earlier. If the SYMBOL TABLE field of a pointer has no value in it, then it is a null reference, and an appropriate message generated.

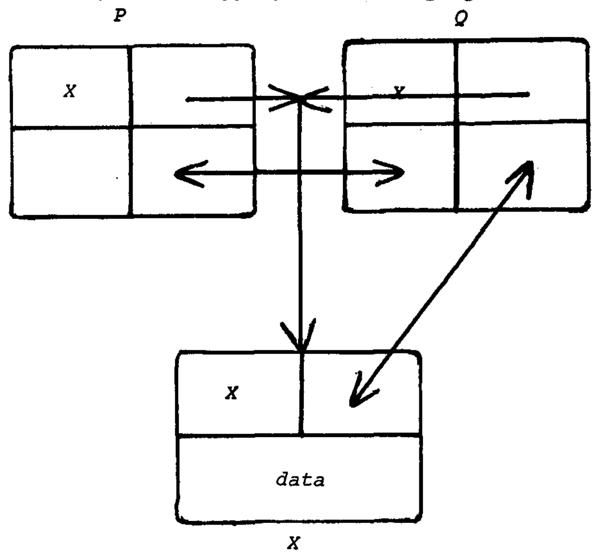


Figure 1. Runtime structures after executing the statements:

```
    ALLOCATE X SET(P);
    Q=P;
```

In all cases, the call to RESOLVE is delayed as long as possible. For example, the left hand side of the assignment statement is resolved after the right hand side expression is evaluated in order to avoid the (remote) possibility of an expression that involves a function call which deallocates the left hand side variable. In addition, based arrays are resolved after all subscripts have been evaluated.

One further problem remained - based references as arguments to a subroutine. In this case, since PL/1 uses a call by reference parameter mechanism, the called procedure does not know that the parameter is based, and may well proceed to deallocate it via its global pointer name, and then reference it via its parameter. In order to prevent this, PLUM assumes that pointers contain storage, and based variables are simply descriptions of this storage. With this interpretation, it is meaningless to pass a based reference to a procedure. If that effect is desired, then the pointer itself can be passed, and the variable referenced via a based variable declared within the procedure.

While this is a restriction from PL/1, we believe that it is reasonable for most applications. In order to accommodate a larger class of programs, however, we make the special case that if a based reference is passed as an argument to a subroutine, then it is placed into temporary storage, and hence passed by value. Thus any deallocation has no effect.

```
PLUM  5:05A   <U. OF MD. PL/1>  06/13/76
      1    P:PROCEDURE OPTIONS(MAIN);
      2       DECLARE A BASED POINTER,
      2               B BASED POINTER,
      2               C BASED POINTER,
      2               P POINTER,
      2               Q POINTER;
      2       /* ALLOCATE C */
      3          ALLOCATE C SET(P);
      3       /* ALLOCATE B */
      4          ALLOCATE B SET(Q);
      5          Q->B=P;
      5       /* ALLOCATE A */
      6          ALLOCATE A SET(P);
      7          P->A=Q;
      7       /* FREE ALL LINKS */
      8          Q=NULL;
      9          P=NULL;
     10       STOP;
     11       END;

COMPILE TIME     158 MSEC.

***** IN    9 ERROR  EX 107 Nothing points
to A any longer. It will be freed.

***** IN    9 ERROR  EX 107 Nothing points
to B any longer. It will be freed.

***** IN    9 ERROR  EX 107 Nothing points
to C any longer. It will be freed.

***** IN  10 EX  24 Program stop
EXECUTION TIME       201 MSEC.
```

Figure 2. Cascading effect of freeing pointers which point to other based pointers.

4. Test Results

After implementing pointers, we were somewhat concerned about the efficiency of the resulting system. While execution time was not of prime importance, we did not want pointers to be so expensive to use that no one would avail themselves of the feature. In addition, since the vast majority of users did not need or even know about pointers, we wanted to keep the overhead to a minimum.

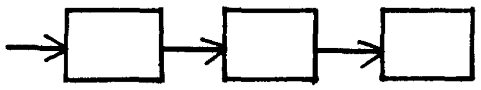
For programs that do not use pointers essentially no overhead was added. Three instructions were added to every procedure activation and termination via the code:

```
SET pointer-field=0;
IF pointers-in-block THEN
     CALL pointer-initialization
```

Since the IF test is almost always false, no additional code needs to be executed. Also, no code is added to the body of a procedure. Thus it is estimated that each program increased in execution time by under 1% - a negligible figure.

In order to test the implementation of pointers, the following typical program was written. A linked list of 10 elements was created, and then deleted (figure 3a). This was repeated 500 times for a total of 5000 allocations. The program implemented the linked lists in two different manners. In one, an array was declared, and the program did its own allocation by having the array element contain the index of the next element in the list. Figure 3b gives the basic code that did the allocation. The variable NEXT is the index of the next array element on the free list while HEAD is the index of the first allocated element. In figure 3c the same basic code is shown using pointer variables. In this case HD points to the head of the list and HD->P points to the next element.

After each allocation, the program was modifed to have 0, 4, 8, 12 and 16 references to an element in the list. Thus the overhead of accessing based references, relative to their allocation, could be monitored.



(a)

```
IF NEXT=0 THEN           DECLARE P BASED POINTER;
     CALL OUTOFSPACE     DECLARE Q POINTER;
NEW=NEXT;                DECLARE HD POINTER;
NEXT=A(NEXT);            ALLOCATE P SET(Q);
A(NEW)=HEAD;             Q->P=HD;
HEAD=NEW;                HD=Q;

    (b)                      (c)
```

Figure 3. (a) List structure created
(b) Allocation using arrays
(c) Allocation using pointers

The results of this test are graphed in figure 4. A total of 4 different programs were tested. Line (a) is the time required with no subscript checking in the array allocation. Line (b) is array allocation with subscript checking turned on. Line (c) is the pointer implementation where type checking was not needed. That is, the pointer was accessed via the same based reference as it was assigned. Finally line (d) is the case where full type checking was needed since the pointer was accessed via a based variable that was different, but had the same attributes. (Note that lines (c) and (d) do not meet at 0 accesses. This is because line (d) still had to type check the based reference on the FREE statement, in order to check for errors in PL/1.)

As expected, the runs without subscript checking took the least time since inline code was generated for subscripts. This code is comparable to the code produced by a production compiler. Similarly the run with full pointer type checking took the most time due to the complex tree walking algorithm that is needed.

What was most surprising were lines (b) and (c). Most users run PLUM with subscript checking turned on. Since PLUM is a diagnostic system, subscript checking is an important diagnostic tool, and from a practical point of view, it is also the default condition, and most users, being lazy, prefer not to modify the defaults unless they run into explicit problems with it. Similary, we believe that for most users of pointers, the based variable used for the assignment to a pointer will be the same based variable used in its access. Thus full type checking is usually unnecessary and only dangling references need to be checked.

With no accesses of based storage between allocations, the pointer implementation took about 50% more time than the array implementation with subscript checking; however, with more than six accesses per allocation, the pointer implementation is more efficient. That is, the overhead to perform subscript checking is more than the overhead to perform pointer validity checking. While we may be able to speed up subscript checking, the validity checking for pointers (without type checking) is inherently a simpler operation.

(Note that further details of the implementation (somewhat dated since it was written before the actual implementation began) as well as how the PL/1 controlled storage class was implemented can be found in [7].)

5. Conclusions

The results of this test seem to indicate the opposite of what we expected. For the majority of programs that will use pointers in PLUM, execution times will be comparable, or even less, than the corresponding program which uses linked arrays. By typing pointers, an efficient algorithm has been implemented which is inherently simpler than the equivalent subscript checking algorithm. While it is possible to use a significant amount of time for pointer type checking (line d of figure 4), our approach is that for the vast majority of users who do not know (or care) about pointers no overhead is

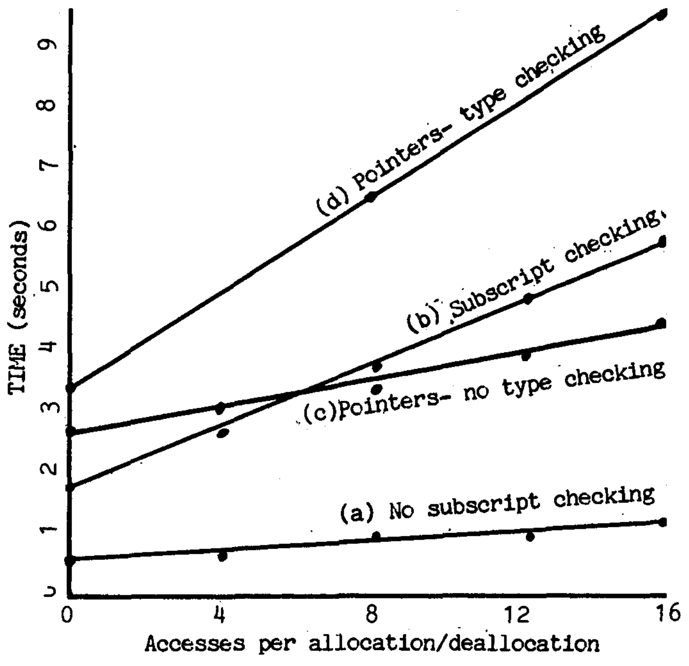


Figure 4. Pointer-Array timing comparison

introduced. For the majority of those who do use pointers, then the implementation is efficient. Finally for those who insist on executing more complex programs than is generally necessary, they will pay in execution time. In any case, error checking will be performed regardless of what they do.

References

[1] Berry D. M., et. al. On the time required for retention. ACM/IEEE Symposium on High Level Language Computer Architecture, SIGPLAN Notices, November, 1973, 165-178.

[2] Chirica L. M., et. al. Two parallel EULER runtime models: the dangling reference, imposter environment and label problems. ACM/IEEE Symposium on High Level Language Computer Architecture, SIGPLAN Notices, November, 1973, 141-151.

[3] Johnston J. B., The contour model of block structured processes. Proc. ACM Symposium on Data Structures in Programming Languages, SIGPLAN Notices, February, 1971, 55-82.

[4] Lindsey C. H. and S. G. van der Meulen, Informal Introduction to Algol 68. North Holland Publishing Company, Amsterdam, 1971.

[5] Lomet D. B., Scheme for invalidating references to freed storage. IBM Journal of Research and Development, January, 1975, 26-35.

[6] Wirth N., The programming language PASCAL. Acta Informatica 1, 1971, 35-63.

[7] Zelkowitz M. V., Pointer variables within a diagnostic compiler. University of Maryland Computer Science Technical Report TR-343, December, 1974.