The following paper was originally published in the
Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)
Monterey, California, June 1995.

Debugging Storage Management Problems in Garbage-Collected Environments

David L. Detlefs and Bill Kalsow
Digital Equipment Corporation Systems Research Center
Palo Alto, CA 94301

Debugging Storage Management Problems in Garbage-Collected Environments

David L. Detlefs and Bill Kalsow
Digital Equipment Corporation
Systems Research Center
Palo Alto, CA 94301
{detlefs,kalsow}@pa.dec.com

May 12, 1995

Abstract

Garbage collection does not solve all storage management problems; programs allocate too much garbage, requiring excess collection, and may retain too much storage, causing heaps to grow too large. This paper discusses these problems and presents tools implemented in the SRC Modula-3 system that help solve them.

1 Introduction

Many garbage collection enthusiasts, present authors included, have presented garbage collection as a panacea for all storage management problems. Like all marketing hype, this is something of an exaggeration. This paper discusses storage management problems that occur in garbage-collected systems, and describes some tools used in SRC Modula-3 [4] [6] that aid in detecting and isolating such problems.

2 Problems with automatic storage management

A garbage collector solves the two classic problems of explicit storage management:

1. *dangling pointers,* where a block of storage is deallocated too early, while pointers to the block are still in use. If the block is reallocated, different parts of the program will be using the same region of memory for different purposes, with disastrous results.

2. *storage leaks*, where blocks of storage are allocated but never deallocated. If this happens repeatedly in a long-running program, the program's memory requirements grow without bound. This problem is the converse of dangling pointers.

Tools like Purify [7] help identify these problems, but further programming is necessary to solve them. However, when garbage collection is used, these problems never occur.

If garbage collection solves these problems, then what could go wrong? More than enough, as we shall see. Excessive allocation may cause overly frequent collection. Moreover, even with collection, the heap may grow too large. There are a variety of causes for surprisingly large heaps: data structures that were designed without an upper bound on their size, references to "dead" heap objects that are hidden behind abstraction boundaries, and references hidden by the underlying compilation and runtime system. We consider each these problems in turn.

2.1 Excessive allocation

If a program allocates a great deal of storage for short-term use, it creates a significant amount of garbage. That garbage must be collected; the more quickly garbage is created, the more often it must be collected. Generational techniques can help greatly in decreasing the cost of collecting such short-lived garbage. However, it is still possible to optimize the performance of most garbage-collected programs by locally reusing storage for the most frequently-allocated types, thereby avoiding garbage-collector overhead. Of course, such techniques have the same dangers as explicit storage management.

2.2 Unbounded data structures

A garbage collector collects storage that is not reachable from the root set (the stacks and global variables) of the program. If the amount of reachable storage increases monotonically over time, a long-running program will still run out of memory, even with garbage collection. It is surprisingly common for programmers of long-running systems to create data structures that grow without bound. For example, programs often use caches to avoid redundant computation. If a program was originally used in a "short-lived" context, that is, it was used to compute a result and then exit, every result may have been cached. If this same program is converted for use in a "long-lived" server context, or is used on much larger input problems, then this strategy is unacceptable; a policy and mechanism for regulating the cache size must be added. This may sound obvious, but if the program is large and is developed by many programmers, it may be difficult to pinpoint all such data structures. Some may occur in unfamiliar libraries, perhaps written by third parties, and perhaps available only in object form.

2.3 References hidden by abstraction

A data structure whose concrete state references a heap object while its abstract state does not may also cause the heap to grow too large. For example, consider the simple stack type whose interface and implementation are shown in figure 1.

The `Pop` procedure removes the top element pointer from the abstract stack. But note that the "removed" pointer remains in the concrete state of the stack; the `elems` array is not modified by `Pop`. So if we pushed 100 pointers to large graph structures, then popped them all, and then didn't use the stack again, the graph structures would be retained as long as the stack was; if the stack were a global variable, this would be for the remainder of the program's lifetime.

This kind of problem can be especially bothersome to pin down, since we are accustomed to thinking of our data types in abstract terms whenever possible. It is therefore necessary in a garbage-collected environment to modify such data structures to keep the references in the abstract and concrete states synchronized. In the example above, we would modify the `Pop` procedure to remove the concrete reference, as shown in figure 2.

2.4 References hidden by the system

A similar problem can occur in places beyond the programmer's control. Consider an execution of a program with procedures, A, B, C, and D. A calls B, whose preamble reserves space on the stack for a local variable x, a pointer to a heap object X. B calls C, but saves on the stack the value of x, which had been in a register, creating the situation shown in figure 3 (part a). C returns, and B returns, leaving the pointer in a dead area of the stack, as shown in figure 3 (part b). At this point there is no problem; if X is otherwise unreferenced, a collection could reclaim it. But A now calls D, which also allocates space on the stack to store a value. Before D stores anything into this location, however, a collection occurs – perhaps D requested a heap allocation. The heap pointer stored by B is dead, but is located in the active area of the stack, as shown in figure 3 (part c). The object X, and all objects reachable from it, will be retained by the collection.

We should note that this problem is probably not too important in single-threaded systems, since any pointer on the stack that causes storage to be retained in one collection is likely be overwritten by stack ac-

```
INTERFACE RefStack;
TYPE
  T <: Public;
  Public = OBJECT
    push(r: REFANY);
    pop(): REFANY;
  END;
END RefStack.



MODULE RefStack;

REVEAL
  T = Public BRANDED OBJECT
    elems: ARRAY [0..99] OF REFANY;
    sp: INTEGER := 0;
   OVERRIDES
    push := Push;
    pop := Pop;
  END;

PROCEDURE Push(self: T; elem: REFANY) =
   BEGIN
     self.elems[self.sp] := elem; INC(self.sp);
   END Push;

PROCEDURE Pop(self: T): REFANY =
   BEGIN
     DEC(self.sp); RETURN self.elems[self.sp];
   END Pop;

BEGIN END RefStack.
```

Figure 1: References hidden by abstraction.

```
PROCEDURE Pop(self: T): REFANY =
  VAR res: REFANY;
  BEGIN
    DEC(self.sp);
    res := self.elems[self.sp];
    self.elems[self.sp] := NIL;
    RETURN res
  END Pop;
```

Figure 2: Corrected version of `Pop`.

tivity before the next collection occurs. However, in multi-threaded environments, the problem may be more serious. Imagine in the example above that procedure D, instead of triggering a garbage collection, waits on a condition variable that is rarely signalled. The thread executing D will be blocked for some time, perhaps for many garbage collections. Those collections will retain the object X.

Note that this is not a problem confined to *conservative collectors* such as the collectors of Bartlett [1] or Boehm and Weiser [2], which assume any bit pattern in the stack that looks like a pointer is a pointer. Lisp systems using hardware tags are just as vulnerable; the pointer values in the stack locations are perfectly valid pointers — they just aren't live at the time of collection. The Boehm-Weiser collector attempts to prevent this problem; it zeros the part of the stack above the stack pointer on each collection. A complete solution to this problem requires a great deal of cooperation between a garbage collector and

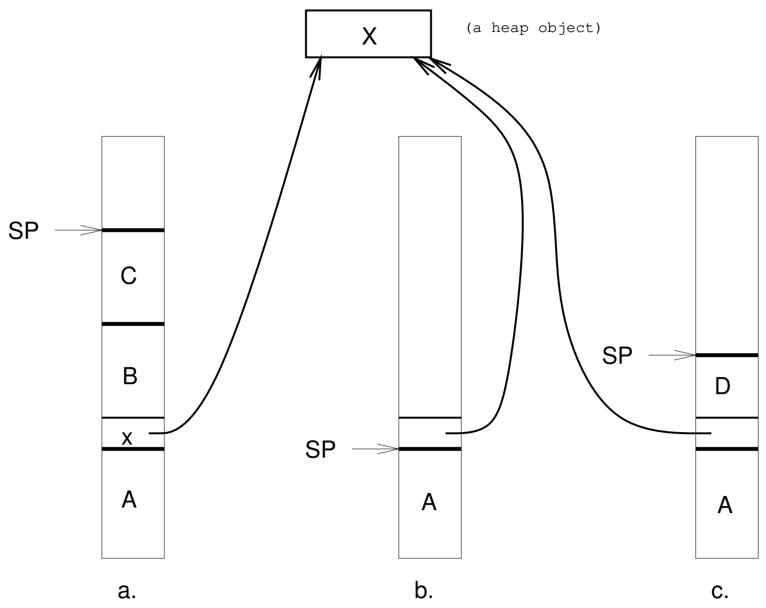

Figure 3: References hidden by the runtime-system.

a compiler. The compiler must either enable the collector to precisely determine which stack values and registers are live at the start of a collection, or generate code to "NIL out" pointer-containing stack locations on procedure entry or exit.

Finally, we should note that while this problem may seem obscure, it does occur in practice. Retention of excess storage was traced to precisely this situation in the first version of the Vesta configuration management system [5]. We know of no certain fix for this problem other than requiring compilers to produce code in which procedures zero-fill their stack frames on entry (or, alternatively, on exit). The obvious performance penalties of these solutions make them somewhat unattractive.

3 Tools

This section describes four tools we have developed to aid programmers both find and fix storage management problems in long-lived garbage-collected programs. These tools are all implemented as part of the runtime system for SRC Modula-3. We give "real-life" examples of the use of each tool. These examples arise from problems encountered in the *Extended Static Checking* (henceforth *ESC*) program verification system being developed at SRC.

3.1 Diagnosing excessive allocation

Excessive allocation causes frequent and potentially intrusive garbage collection. *Shownew* is a tool that allows a user to observe the allocation behavior of a program. Shownew is integrated into the runtime system, so that any SRC Modula-3 program can be passed a special command-line argument that will cause it to run under the control of a Shownew process. Shownew presents a bar graph indicating how much storage of each type is being allocated. A menu allows the user to indicate whether the graph should display the number of objects or bytes allocated, and whether the numbers should indicate totals since the beginning of the program, or only new allocations since the display was last updated.

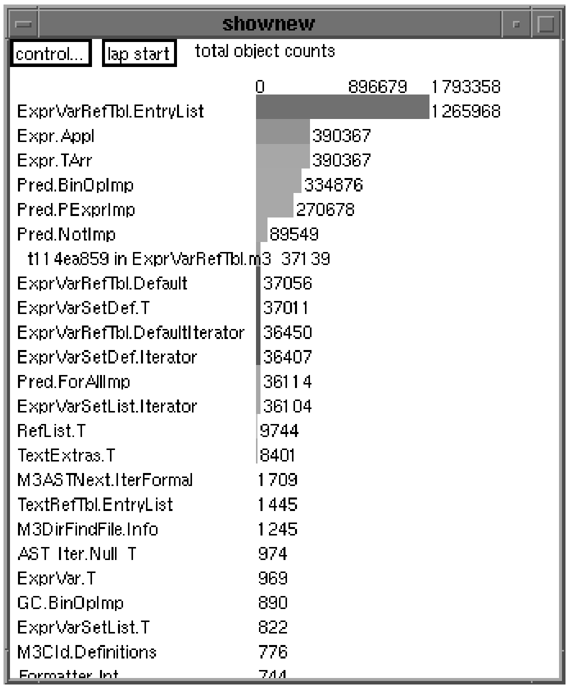

Figure 4: Shownew output (before).

Shownew allows the programmer to pinpoint what types are being allocated most often and might be causing excessive collection.

With Shownew, it is quite easy and almost always worthwhile to determine what types are allocated most frequently in your program. The answers are sometimes surprising, and occasionally represent bugs that are simple to correct. When the ESC system was found to be embarrassingly slow on a new example, we tried Shownew to see if excessive allocation was a problem. The result is shown in Figure 4. (Of course, the bars of the graph appear in color on a color monitor.) The identity of the type allocated most frequently was a complete surprise. `ExprVarRefTbl.EntryList` is an internal type used in the implementation of a set type provided by the Modula-3 library. These allocations were eventually traced to code that applied a variable substitution to a universally quantified formula (a `ForAll`), as shown in figure 5. The argument `exc` is a set of "excluded" variables, variables that are *not* to be substituted. Since a quantifier binds the quantified variables, `ForAllSubst` adds its own quantified variables, `self.vs`, to `exc` before applying the substitution to its body.

It happened that the type `ExprVarSet.T` was implemented using hash tables. The set type's `union` method is non-destructive, so `exc.union(self.vs)` makes a copy of `exc`, adds the elements of `self.vs`, and returns the copy. Note that the copy is discarded as soon as the call to `MkForAll` returns.

We modified this procedure to avoid the copy, as shown in figure 6. The usual way to avoid this copy, and the one we used, is to substitute destructive operations (in which `exc` is modified) for functional operations (in which `exc` is not modified). The call `exc.unionD(self.vs)`, for example, adds the elements of `self.vs` to the set `exc`, modifying its value. Similarly, `diffD` destructively deletes elements from a set.

In this example, several assumptions are necessary to argue the correctness of the transformation. First, `exc` is not being accessed by any concurrently executing threads. Second, it is a precondition of `ForAllSubst` that `exc` and `self.vs` are disjoint. Third, the call `self.body.subst(s, exc)` does not retain a reference to `exc`. Using destructive operations often yields significant performance benefits, but the subtlety of the assumptions necessary to show them correct argues that they should be used sparingly. A tool like Shownew helps identify the most promising targets.

The first picture in figure 7 shows the result of Shownew on this program after the change described above. Note that the `EntryList` type is now only the sixth of the most frequently allocated types; the number of `EntryLists` allocated decreased by more than 95%. Similar transformations resulted in the situation shown in the second picture of that figure, where the absolute numbers of the most frequently allocated types have dropped dramatically.

The implementation of Shownew is fairly simple. Each (garbage-collected) heap-allocated object in Modula-3 has a header that contains a *typecode*, a unique integer corresponding to the dynamic type of the object. From a typecode it is easy to find the name, size, and various other attributes of the corresponding type. A `NEW(T)` expression in the source is compiled to a call to the runtime's allocation routine with the typecode of `T` as an argument. When

```
PROCEDURE ForAllSubst(self: ForAll; s: Subst.T;
                      exc: ExprVarSet.T): ForAll =
  BEGIN
    RETURN MkForAll(self.vs, self.body.subst(s, exc.union(self.vs)));
  END ForAllSubst;
```

Figure 5: Excessive allocation (before).

```
PROCEDURE ForAllSubst(self: ForAll; s: Subst.T;
                      exc: ExprVarSet.T): ForAll =
  VAR res: ForAll;
  BEGIN
    exc := exc.unionD(self.vs);
    res := MkForAll(self.vs, self.body.subst(s, exc));
    exc := exc.diffD(self.vs);
    RETURN res;
  END ForAllSubst;
```

Figure 6: Excessive allocation (after).

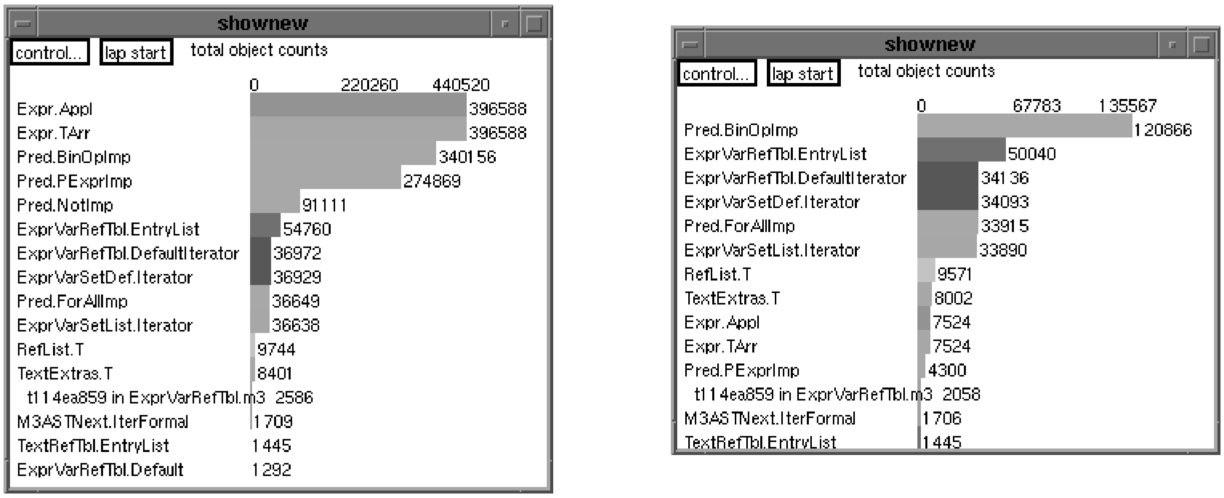

Figure 7: Shownew output (after)

Shownew is being used, the allocator counts the objects allocated of each type, periodically forwarding that information to the Shownew process.

3.2 Excess retained storage

The previous section dealt with allocation. Allocation and heap size are obviously related, since something must be allocated to become part of the heap, but they are also quite different. A type that is allocated often may contribute nothing to the heap size, if all instances quickly become garbage and are collected, while a type that is allocated infrequently, but whose instances are retained, will cause the heap to grow without bound.

The rest of this section introduces three tools used to diagnose problems in the theorem-prover used in ESC. In long-running proofs, the theorem-prover's heap seemed to be growing continually larger, while we expected the storage requirements of the systems to quickly reach a steady state.

3.2.1 RTutils.Heap

The first tool we used was the procedure `RTutils.Heap`, which reports the composition of the heap by data type. If we explicitly run a collection before generating such a report, we obtain a breakdown by type of live objects in the heap. RTutils.Heap has several options that allow the report to be ordered by number of allocated objects or bytes occupied, and to limit the report to the top n types by the requested ranking method. We modified ESC to periodically perform a garbage collection and call RTutils.Heap, reporting the top 10 types ranked by bytes occupied. Figure 8 shows these reports near the beginning (but after we would have expected the heap size to stabilize) and end of an execution of the theorem-prover on a relatively small test case.

The heap has grown by almost three megabytes. The bulk of that increase can be attributed to the type Enode.Parent, which grew by more than two megabytes.

The addition of calls to RTutils.Heap to the source code has helped us identify the first-order term of our problem: we are retaining more Enode.Parent objects than we expected. These reports are useful enough that we left this debugging code in permanently, with printing of the reports controlled by an environment variable.

The implementation of RTutils.Heap requires the ability to enumerate all the objects in the heap and classify them by type. Recall that Modula-3 heap objects are prefixed by headers containing a typecode that uniquely designates a type. It is therefore simple to examine all heap objects, constructing a table mapping typecodes to object counts and bytes occupied. Finally, the table is sorted by the appropriate metric and printed.

Two additional features of RTutils.Heap merit explanation.

First, sometimes a report by typecode is too coarse-grained. For some problems, it is convenient to program in a Lisp-like style within Modula-3. Instances of the type RefList.T are singly-linked lists of generic pointers. This type can be used much like a Lisp cons cell. In programs that use such a style, lists are used in a number of different ways. A report saying that the heap contains many RefList.Ts may not identify which of the many uses of the type is responsible for the retained storage. In such a situation, the programmer can use the RTAllocStats interface. This interface exports a procedure EnableTrace, which takes a typecode argument. Once EnableTrace is called for a type, the header of each newly allocated object with that type is annotated with a small integer representing the *call site* of the allocation. Abstractly, a call site can be thought of as a snapshot of the stack at the time of the allocation. Practically, it is the sequence of the top n program counter values on the stack, where n defaults to 3 but can be set by the user. The single program counter of the actual allocation is insufficient; in the RefList case mentioned above, few RefList.Ts are allocated directly by clients of the interface; most are allocated using convenience procedures of the interface like RefList.Cons or RefList.List3. It would be of little use to discover that most lists were allocated within calls to Cons. Figure 9 shows an excerpt from one of these finer-grained reports.

Call-site tracking requires more from the runtime system. The allocation routine must first determine whether tracking is enabled for the type being allocated. If so, it must determine the call site, which requires the ability to interpret a thread stack's contents at runtime; note that this code is highly platform-specific. Because we use "unused" bits in the standard object header to record call sites information, we are currently limited to 256 sites per type. If this limit is reached, allocations at other sites are credited to a site code reserved for other sites, as shown in figure 9.

The second additional feature of RTutils solves the opposite problem: when reports by typecode are too fine-grained. For example, ESC processes Modula-3 using M3TK [3], which translates the source code into an abstract syntax tree (AST) that ESC then analyzes. These AST's are made up of many different types; it would be tedious to determine how much space is occupied by such trees from the kind of report described above. However, the types of the tree nodes are organized in a type hierarchy rooted at a generic AST.NODE type. RTutils.Heap produces a second report for Modula-3 object types that reflects the inheritance hierarchy. Since Modula-3 supports only single inheritance, this hierarchy is a simple tree. In figure 10 it becomes clear that AST nodes occupy about 1.2 megabytes, with different subtypes accounting for different fractions of that total.

The restriction to single inheritance makes the implementation of this report simple. Typecodes are

Near beginning:

```
Code   Count    TotalSize  AvgSize  Name
---- --------- --------- --------- ---------------
 270         1   2457616   2457616 Enode.UndoStack
 180      3352    616768       184 Enode.Parent
 173     14623    350952        24 RefList.T
 230         1    160024    160024 Simplex.RatArr2
 294      3804    121728        32 SigTab.EntryList
 233         4     81984     20496 <anon type>
 143      1857     59424        32 Context.Literal
 178       411     59184       144 Enode.Leaf
 161       473     49192       104 MatchingRule.Rule
 196        19     48216      2537 RTHooks.CharBuffer
     --------- ---------
         33714   4505664
```

Near end:

```
Code   Count    TotalSize  AvgSize  Name
---- --------- --------- --------- ---------------
 180     16395   3016680       184 Enode.Parent
 270         1   2457616   2457616 Enode.UndoStack
 294     14542    465344        32 SigTab.EntryList
 173     14611    350664        24 RefList.T
 233         4    311360     77840 <anon type>
 230         1    160024    160024 Simplex.RatArr2
 178       852    122688       144 Enode.Leaf
 232         1     64016     64016 Context.TritArray
 143      1790     57280        32 Context.Literal
 161       503     52312       104 MatchingRule.Rule
     --------- ---------
         59951   7684096
```

Figure 8: Use of `RTutils.Heap`

```
Code   Count    TotalSize  AvgSize  Name
---- --------- --------- --------- -------------------------
272          1   2457616   2457616 Enode.UndoStack
173      15416    369984        24 RefList.T
          3436     82464        24    Cons + 16_3c in RefList.m3
                                      SubWork + 16_59c in PredSx.m3
                                      SubWork + 16_4dc in PredSx.m3
          2333     55992        24    Cons + 16_3c in RefList.m3
                                      CopySx + 16_130 in Clause.m3
                                      CopySx + 16_c4 in Clause.m3
...
           465     11160        24    OTHER SITES
...
```

Figure 9: `RTutils.Heap` output broken down by call site.

```
Code   Count    TotalSize  AvgSize  Name
---- --------- --------- --------- -------------------------
...
311      24866   1223512        49     AST.NODE
...
510        964     49208        51           M3AST_AS.STM
...
516        254     12192        48               M3AST_AS_F.Assign_st
...
527         39      2808        72               M3AST_AS_F.For_st
...
```

Figure 10: `RTutils.Heap` output broken down by the type hierarchy.

assigned to types in a pre-order traversal of the type forest, so that the typecodes of the subtypes of a type form a consecutive sequence of integers. The subtype test is therefore a simple range test, making it easy to aggregate the numbers for a type and all its subtypes.

In summary, `RTutils.Heap` answers some of the most basic questions that must be answered when debugging a storage management problem: how big is the heap, what kind of objects are in it, and how many objects of each type are in it? It can answer these questions at finer or coarser grains where necessary.

3.2.2 RTHeapStats.ReportReachable

`RTutils.Heap` identified *what* was filling up the heap, but did not tell us *why* the objects filling the heap were escaping collection. The next tool in our kit helps answer that question. An object is retained by garbage collection only if it is reachable from a *root* of the program, that is, from the stacks, registers, or global variables. The procedure `RTHeapStats.ReportReachable` tells us how many bytes of storage are reachable from the roots, breaking down the roots in various ways. Each global variable in Modula-3 is associated with some module. One list ranks the modules by bytes reachable from their global variables. A more detailed breakdown ranks the individual global variables by the amount of storage they reach. A second section details storage reachable from thread stacks, again, reporting both at coarse and fine grains. In the coarse grained report, entire stacks are lumped together. In the fine grained report, stack frames are printed along with the storage reachable from individual references contained in those frames.

We modified ESC to call `ReportReachable` periodically. Figure 11 shows excerpts from two reports, one near the beginning of the program, but after we expected the heap size to reach a steady state, and one from near the end.

Comparing the reports identifies the `Enode` module as the major offender. We see also that an object of type `SigTab.Default` in the `Enode` module went from reaching just over one megabyte to almost three and one half. The only global variable of this type in the module was named `sigTab`. Once we had identified `sigTab` as a possible problem, a moment's thought was sufficient to reveal our mistake. The purpose of `sigTab` is to ensure that we never create distinct `Enode.Parents` with identical children. For small examples it was acceptable to remember all the parents ever created. For large examples, however, such unbounded growth leads to overly large heaps. This table is a cache of parent nodes; we needed to invent a cache-management policy for it. We can delete a `Parent` from the table when it is no longer in use. Because of its backtracking search structure, the prover "unwinds" every action it takes; so we can delete a `Parent` from the table when we undo the action that added it in the first place.

We now briefly describe the implementation of `ReportReachable` before continuing with our detective story. Essentially, each line item in the report is the result of a mini-garbage-collection. We allocate a new bit vector large enough to serve as mark bits for all the objects in the heap. To determine the set of objects reachable from some subset r of the roots, we clear the mark bits, then (recursively) mark each object reachable from r, using the mark bits to terminate the recursion. When the mark phase is complete, we count the objects and bytes corresponding to the mark bits. This algorithm requires the ability to enumerate the pointer-containing fields of an arbitrary heap object. Note also that presenting the result of thread stack traversals requires the same run-time interpretation of thread stacks needed for call-site tracking in section 3.2.1.

Note that this process is potentially quite expensive. If a heap contains a large linked data structure that is reachable from many roots, that structure will be traversed and counted once for each of the different root references. In practice, however, the performance of `ReportReachable` seems quite acceptable; the reports shown above did not slow down the the execution of the program greatly.

3.2.3 RTHeapDebug

In the ESC problem we have been discussing, it was fairly easy to identify a problem once `RTHeapStats.ReportReachable` led us to an offending global variable. Sometimes, however, this information may not be sufficient; it may be difficult to determine what paths exist from the offending root to objects of the problematic type. The `RTHeapDebug` interface helps find such paths.

Even in a garbage-collected system, it is sometimes possible to identify a point in the program where one expects an object to become garbage. In our ongoing example, we added code to remove an `Enode.Parent` from `sigTab` when the prover's search backtracked past the point where the parent was first created and added to the table; we expected the parent to be unreachable after it was deleted from the table. `RTHeapDebug` allows the programmer to check such expectations.

Calling `RTHeapDebug.Free(r)` asserts that the reference `r` is unreachable. A call to `RTHeapDebug.CheckHeap` does the equivalent of a garbage-collection, traversing all reachable objects in search of any that have been asserted unreachable. If such an object is found, `CheckHeap` prints a path from a root to the putatively unreachable object.

In the ESC example, we called `RTHeapDebug.Free` to assert that `Enode.Parents` removed from `sigTab` by the undo code were unreachable. We called `CheckHeap` at a later point. We were surprised to find that our assumption was wrong. Figure 12 shows an excerpt of a report from

Near beginning:

```
HEAP: 0x14009a000 .. 0x140c98000 => 11.9 Mbytes

Module globals:
 # objects   # bytes  unit
 ---------  --------  -----------------
     24471   4151816  Enode.m3
     11116    698648  Context.m3
      8390    579608  Clause.m3
       ...

Global variable roots:
 # objects   # bytes         ref type                 location
 ---------  --------  ---------- -----------------   ------------------
      8450   3036328  0x140694008 Enode.UndoStack     Enode.m3 + 2224
      9065   1098920  0x1402a2558 SigTab.Default      Enode.m3 + 1352
     12925    703144  0x1400a7990 IntRefTbl.Default   Enode.m3 + 4328
       ...
```

Near end:

```
HEAP: 0x14009a000 .. 0x141244000 => 17.6 Mbytes

Module globals:
 # objects   # bytes  unit
 ---------  --------  -----------------
     43159   6534144  Enode.m3
     11194    757936  Context.m3
      5215    748008  Simplex.m3
       ...

Global variable roots:
 # objects   # bytes         ref type                 location
 ---------  --------  ---------- -----------------   ------------------
     27149   3456264  0x1402a2558 SigTab.Default      Enode.m3 + 1352
      8632   3100256  0x140694008 Enode.UndoStack     Enode.m3 + 2224
     13471    727832  0x1400a7990 IntRefTbl.Default   Enode.m3 + 4328
       ...
```

Figure 11: Use of `RTHeapStats.ReportReachable`.

```
Path to 'free' object:
   Ref in root at address 0x14000a770...
   Object of type Enode.UndoStack at address 0x14013c008...
   Free object of type Enode.Parent at address 0x14013eac8...
Path to 'free' object:
   Ref in root at address 0x14000a770...
   Object of type Enode.UndoStack at address 0x14013c008...
   Free object of type Enode.Parent at address 0x14013eae0...
Path to 'free' object:
   Ref in root at address 0x14000a770...
   Object of type Enode.UndoStack at address 0x14013c008...
   Free object of type Enode.Parent at address 0x14013eaf8...
...
```

Figure 12: Use of `RTHeapDebug.CheckHeap`.

`CheckHeap`. There was only one variable in the program of type `Enode.UndoStack`; it is the stack on which undo records are written to implement the backtracking search. Apparently this stack was pointing directly to objects we thought should be unreachable.

A little investigation revealed the problem to be a case of references hidden by abstraction, almost exactly the situation described in the example of section 2.3. We considered the undo stack as only the portion below the stack pointer, but the garbage collector considered the entire array data structure. Undo records above the stack pointer contained pointers to otherwise unreachable objects. To solve the problem we modified the *pop* operations of the various undo stacks in the system to set pointers in dead stack entries to `NIL`. This investigation has purged ESC of its most egregious storage leaks.

It might be argued that the technique embodied by `RTHeapDebug` is regressive, in that it requires the programmer to do the equivalent of explicit storage management. There is some truth to this argument. However, `RTHeapDebug.Free` is needed only for types identified as storage problems, not for all deallocated objects. Calling `RTHeapDebug.Free` for an object that is still accessible causes a graceful error message, while actually deallocating a still-accessible object is likely to cause a confusing dangling pointer bug.

The implementation of `RTHeapDebug` uses the `WeakRef` interface provided by the Modula-3 runtime. A `WeakRef.T` is a reference to an object that does not prevent the object from being collected. `WeakRef` provides a routine for obtaining the "real" reference corresponding to a `WeakRef.T`; this routine returns `NIL` if the referenced object has been collected. `RTHeapDebug` maintains a set of weak references to freed objects, initially empty. `RTHeapDebug.Free(r)` adds a weak reference corresponding to `r` to this set. `RTHeapDebug.CheckHeap` first throws away any elements in the set whose referents have been collected, then searches for paths to the remaining freed objects.

4 Conclusion

We present four classes of storage-management problems that occur even in completely garbage-collected systems: excessive allocation, data structures that grow without bound, references hidden by abstraction boundaries, and references hidden by the system. We have presented tools that aid the diagnosis of such problems. These tools are part of the SRC Modula-3 system, but could easily be adapted for other languages; in fact, some of the tools are not specific to garbage-collected systems.

We feel that the trend towards long-lived server programs will speed the adoption of garbage-collected systems; the likelihood of pointer-smashes or storage leaks in explicitly managed systems is far too high for applications that must run reliably for long periods. This paper addresses the (more manageable!) storage management problems that remain once such systems have adopted garbage collection.

References

[1] Joel F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, February 1988.

[2] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[3] Mick Jordan. An Extensible Programming Environment for Modula-3. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*. ACM, ACM Press, 1990.

[4] Bill Kalsow. SRC Modula-3 home page. URL http://www.research.digital.com/SRC/modula-3/html/home.html.

[5] Roy Levin and Paul Mcjones. The Vesta Approach to Precise Configuration in Large Software Systems. Research Report 105, Digital Equipment Corporation Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, June 1993.

[6] Greg Nelson, editor. *Systems Programming in Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[7] Pure Software, Los Altos, California. *Purify Version 1.1 Beta A*, 1992. User manual.