Copying Garbage Collection in the Presence of Ambiguous References

Andrew W. Appel and David R. Hanson
Department of Computer Science, Princeton University,
Princeton, New Jersey 08544

Research Report CS-TR-162-88

June 1988

ABSTRACT

Garbage collection algorithms rely on invariants to permit the identification of pointers and to correctly locate accessible objects. These invariants translate into constraints on object layouts and programming conventions governing pointer use. There are recent variations of collectors in which the invariants are relaxed. Typically, rules governing pointer use are relaxed, and a 'conservative' collection algorithm that treats all potential pointers as valid is used. Such pointers are 'ambiguous' because integers and other data can masquerade as pointers. Ambiguous pointers cannot be modified and hence the objects they reference cannot be moved. Consequently, conservative collectors are based on mark-and-sweep algorithms. Copying algorithms, while more efficient, have not been used because they move objects and adjust pointers. This paper describes a variation of a copying garbage collector that can be used in the presence of ambiguous references. The algorithm constrains the layout and placement of objects, but not the location of referencing pointers. It simply avoids copying objects that are referenced directly by ambiguous pointers, reclaiming their storage on a subsequent collection when they are no longer ambiguously referenced. An implementation written in the ANSI C programming language is given.

Copying Garbage Collection in the Presence of Ambiguous References

Introduction

Traditional garbage collection algorithms [1] are based on the assumption that memory use obeys well-defined rules. Such rules typically specify the layout of objects allocated in the heap, the use of 'root' or 'tended' pointers, and the representation of data. These kinds of rules are designed to maintain invariants necessary for successful garbage collection. Rules concerning root pointers enforce invariants that permit all active objects to be located starting from a (small) set of root pointers. Rules concerning object layout and data representation permit pointers to objects to be distinguished from other data.

These kinds of rules exert a strong influence on the design and implementation of systems that use garbage collection for storage management. Programming languages, such as Icon [2] and ML [3], are prime examples, but garbage collection constraints affect the design of other applications, such as editors [4].

Evolving systems that lack garbage collectors initially may suffer because adding garbage collection late in the development can be prohibitively difficult. Adding garbage collection to an existing or nearly completed system often requires adherence to rules that can lead to massive re-design and re-implementation. Bugs in a garbage collector that has not matured concurrently with its host system are notoriously difficult to repair [5].

Garbage collection is sufficiently useful that several attempts have been made to provide garbage collectors that can be used in systems *without* the accompanying rules and consequent impact outlined above. Such garbage collectors must deal with ill-defined root sets and *ambiguous* references, i.e., apparent references to objects that cannot be distinguished from other data, such as integers. In the most general case, these references arise from data in the root set and from within other objects.

The approach usually taken is 'conservative': All global data including the runtime stack is treated as the root set, and any object that is potentially accessible is treated as active. Thus, any bit pattern in the root set or in an accessible object that, when treated as a pointer, points to an object marks that object as active.

While numerous such facilities have undoubtedly been implemented, few are documented. The `galloc` facility in the ninth edition of UNIX is an example of this conservative approach [6]. `galloc` and `gfree` are generalizations of the `malloc` and `free` facility from the C programming language library [7]. The latter functions allocate and deallocate storage explicitly, but do not reclaim storage. `galloc` and `gfree` are similar, except that inaccessible space is reclaimed via garbage collection when free space is exhausted.

More recent facilities are similar [8, 9] and take special efforts to minimize the additional overhead incurred by programs that do *not* make use of the garbage collector. These recent efforts have also demonstrated the use of the garbage collector as a debugging tool. Adding a garbage collector to an existing system in which storage is supposedly managed explicitly can often help detect 'storage leaks' and similar storage management bugs.

Some garbage collection algorithms move accessible objects and adjust pointers to them to reflect this movement. In the presence of ambiguous references, however, pointers cannot be adjusted because they cannot be identified conclusively as pointers. Thus, objects accessible via ambiguous references cannot be moved. For this reason, collectors like `galloc` are based on mark-and-sweep algorithms with an associated free space list [1].

Copying garbage collection algorithms [10] are generally superior to mark-and-sweep. Copying algorithms require time proportional to the number of accessible objects whereas mark-and-sweep algorithms require time proportional to the number of accessible *and* inaccessible objects. With the increasing prevalence of large memories, the additional space required by copying algorithms is of decreasing importance [11].

In general, copying algorithms cannot be used with ambiguous references because accessible objects are moved [12–14]. The general situation does not always apply, and in some cases, the details of specific systems relax some of the general constraints and admit more flexibility. The remainder of this paper describes a simple copying garbage collector that *can* be used when there are relatively few ambiguous references and when those references appear only in the root set. As detailed below, the collector simply declines to move objects directly accessible from ambiguous references, reclaiming their storage on a subsequent collection when they are no longer referenced by ambiguous pointers. The collector is written in the ANSI C programming language [7].

The algorithm described below is less general than the one described recently by Bartlett [12, 13]. Bartlett's algorithm handles 'derived pointers' and ambiguous pointers from within the heap. It does, however, require that pages with ambiguously referenced objects remain allocated until none of the objects on the page are ambiguously referenced.

Environment

Systems in which space is allocated without any knowledge of the contents are rare in practice. Most often, dynamic memory allocation is used with specific data structures having known layouts. For example, compilers often use general-purpose storage allocators, but usually allocate relatively few different types of objects, such as symbol table entries and tree nodes. In some object-oriented systems, such as Objective-C [15], may have many different kinds of objects, but they obey simplifying rules and conventions. When the details of the allocated objects are known, general-purpose techniques can often be simplified.

The copying garbage collector described below is intended for use in systems in which dynamically allocated objects are used as described above. Specifically, it is assumed that

(1) the layout of objects is known;
(2) objects are self-identifying;
(3) the memory spaces from which objects are allocated contain only objects;
(4) all pointers to an object point to the head of an object, i.e., there are no 'derived pointers'; and
(5) objects do not contain untagged 'unions' (i.e., variant records that can hide pointers).

These assumptions constrain the layout and placement of objects, but they do not constrain the placement of pointers from outside the allocation space. The garbage collector treats any pointer to an object as valid and preserves the referenced object.

Objects are defined by

```
typedef struct object Object;
struct object {         /* generic object: */
        int     size;           /* number of fields */
        Object  *alink;         /* link for ambiguous object list */
        Object  *fields[1];     /* pointers to other objects */
};
#define sizeInBytes(n) (((n)-1)*sizeof(Object *) + sizeof(Object))
```

Instances of `Object` are as large as necessary to accommodate their fields. `sizeInBytes` computes the size of an object in bytes given its size in fields. The `size` field is unnecessary where the size can be deduced from other fields [2]. Also, assuming that all fields are pointers simplifies the exposition; in applications involving heterogenous objects, a pointer map would be used to find pointers within objects. Pointer maps, one per object type, are generated by hand or by the compiler. `alink` is described below.

Memory is divided into two spaces defined by

```
typedef struct space {  /* memory space: */
        char    *start;         /* start of memory space */
        char    *end;           /* 1 past end of memory space */
        char    *next;          /* next free location */
        Object  *limit;         /* 1 past end of free chunk */
        Object  *alist;         /* ambiguous object list */
} Space;
Space A, B, *old = &A, *new = &B;
```

`A` and `B` are initialized to `n`-byte memory regions during initialization, `start` and `end` give the bounds of a space, and the allocation pointers `next` and `limit` and `alist` are described below.

Allocation and Garbage Collection

New objects are allocated and cleared by calling **newObject(n)** where **n** is the number of fields required ($n > 0$):

```
Object *newObject(int n) {
        Object *p;

        p = (Object *) alloc(sizeInBytes(n), old);
        if (p == 0) {
                gc();
                p = (Object *) alloc(sizeInBytes(n), old);
                if (p == 0)
                        error("memory overflow");
        }
        p->alink = 0;
        p->size = n;
        while (--n >= 0)
                p->fields[n] = 0;
        return p;
}
```

alloc(int n, Space *s), described below, allocates **n** bytes in the memory space pointed to by **s**.

As shown, objects are allocated in **old** space. When an allocation fails, indicated by **alloc** returning 0, the garbage collector, **gc**, is called. In the absence of ambiguous references, **gc** would use the standard copying algorithm: Copy accessible objects in **old** to **new** space, exchange **old** and **new**, and return. If the second call to **alloc** fails, the system is out of memory.

gc cannot move objects pointed to by ambiguous references. It leaves ambiguously referenced objects behind, but adds them to the *alist* — the ambiguously referenced object list — for that memory space. The alist for space **s** begins at **s->alist** and is threaded through the **alink** fields of objects in order of increasing object addresses. **s->alist** is initialized to **s->end**, which permits non-zero **alink** fields to indicate an object's presence on a alist. Objects are added to **old->alist** by **aref**:

```
void aref(Object *p) {
        Object **q, *r;

        if (inSpace(p, old) && p->alink == 0) {
                q = &old->alist;
                for (r = *q; p > r; r = *q)
                        q = &r->alink;
                p->alink = r;
                *q = p;
        }
}
```

inSpace(Object *p, Space *s) is non-zero if **p** points to an object in space **s** and is implemented by checking that **p** is within **s->start** and **s->end** and points to the head of an object. Note that the length of the alist at each garbage collection gives the number of ambiguously referenced objects.

The root set consists of the active portion runtime stack, **stack[0..sp]**. Other locations, such as the machine registers and global data, could also be added. **gc** considers all references from the root set to be ambiguous, adding them to **old**'s alist by calling **aref**:

```
gc(void) {
        int i;
        Space *t;
        Object *p;

        old->alist = (Object *) old->end;
        for (i = 0; i <= sp; i++)
                aref(stack[i]);
        old->limit = old->alist;
        head = 0;
        tail = &head;
        for (p = old->alist; (char *)p != old->end; p = p->alink)
                scan(p);
        for (p = new->alist; (char *)p != new->end; p = p->alink)
                scan(p);
        while (head) {
                p = head;
                head = head->alink;
                if (head == 0)
                        tail = &head;
                p->alink = 0;
                scan(p);
        }
        t = old;
        old = new;
        new = t;
        new->next = new->start;
}
```

old->limit is reset to **old->alist** for use in **alloc**, described below. Once **old->alist** is constructed, those objects are treated as the root set for the remainder of the collection. That is, objects pointed to by objects on the alist are *copied* to the **new** space using **scan**. Similar comments apply for those objects on **new->alist**, which are ambiguously referenced objects left over from the previous collection. This top-level copying is accomplished by the second and third **for** loops in **gc**.

scan moves the object's referents using **move**:

```
void scan(Object *p) {
        int i;

        for (i = 0; i < p->size; i++)
                p->fields[i] = move(p->fields[i]);
}
```

Whenever **move** copies an object to **new** space, it appends the copy onto a queue of unscanned objects. This queue is formed by linking through the **alink** fields, which are not as yet used in the newly copied objects. **head** points to the first object on the queue and **tail** points to link at the end of the queue. The **while** loop in **gc** is the equivalent of the breadth-first code from the standard copying algorithm; it removes and scans objects from the unscanned queue, copying their referents by calling **move**.

The unscanned queue is unnecessary a standard copying collector; the breadth-first loop advances a pointer toward **new->next**, scanning intervening, recently copied objects, e.g.,

```
        for (unscanned = new->start; unscanned < new->next; ) {
                scan((Object *) unscanned);
                unscanned += sizeInBytes(p->size);
        }
```

This code cannot be used in the present collector because the free space in **new** is not contiguous; ambiguously referenced objects left over from a previous collection fragment the free space in **new**.

move copies a unambiguously referenced object from **old** to **new** space, marks it as moved, leaves a forwarding pointer in its **old** place, and adds it to the unscanned queue:

```
Object *move(Object *p) {
        int i;
        Object *q;

        if (!inSpace(p, old) || p->alink)
                return p;
        if ((p->size&MARKED) == 0) {
                q = (Object *) alloc(sizeInBytes(p->size), new);
                if (q == 0)
                        error("memory overflow during collection");
                q->alink = 0;
                q->size = p->size;
                for (i = 0; i < p->size; i++)
                        q->fields[i] = p->fields[i];
                p->fields[0] = q;
                p->size |= MARKED;
                *tail = q;
                tail = &q->alink;
        }
        return p->fields[0];
}
```

MARKED is the sign bit of the size field. If p->alink is non-zero, p points to an ambiguously referenced object, which is not moved. move is similar to the code used in the standard copying algorithm [1]. The differences are the test for a ambiguously referenced object, the possibility that alloc will fail, and the use of a mark bit. In the standard algorithm, testing if fields[0] points into new space was sufficient to determine if the object needed to be copied. Here, however, fields[0] might point to an object that was previously referenced by ambiguous pointer and was left behind in new space on a previous collection, which necessitates the mark bit.

Suppose old and new are initialized as shown above to spaces A and B. At the end of the first call to gc, all accessible unambiguously referenced objects are in B, A contains the (few) ambiguously referenced objects on its alist, and old and new refer to B and A, respectively. When B runs out of memory, gc is invoked, builds an alist for B and begins to move unambiguously referenced objects into A, calling alloc for space. alloc uses A's alist to step over ambiguously referenced objects left behind during the first call to gc:

```
char *alloc(int n, Space *s) {
        for (;;) {
                if (s->next + n < (char *)s->limit) {
                        s->next += n;
                        return s->next - n;
                }
                if ((char *)s->limit >= s->end)
                        return 0;
                s->next = (char *)s->limit + sizeInBytes(s->limit->size);
                s->limit = s->limit->alink;
        }
}
```

At any point during execution, s->next and s->limit delimit the next free portion of space s. If an allocation request cannot be satisfied, s->next is stepped over the intervening ambiguously referenced object and s->limit is advanced to the next object on the alist. As noted above, the last object on the alist points to s->end.

At the end of the second call to gc, all accessible unambiguously referenced objects have again been copied, filling in the holes in A, ambiguously referenced objects have been left behind in B on its alist, and old and new refer to A and B, respectively. Allocation continues in A, using alloc to step over ambiguously objects left over from the first gc, as necessary. When space in A is exhausted, a third call to gc continues the cycle. A's alist is rebuilt, potentially releasing the space occupied by ambiguously referenced objects found during the first collection, B's alist is scanned, and unambiguously referenced objects are copied to B.

Improvements

One possible problem with the garbage collection algorithm is fragmentation. If an allocation request cannot be satisfied by alloc, the space between s->next and s->limit is lost. More complex allocation schemes can be used to avoid this problem. For example, every time alloc steps over some free space, it could add it to a free space list. alloc would search the free space list only when its other, faster, allocation method failed, or when success was certain, which can be determined by keeping track of the maximum block size on the free list. This scheme would add another two fields to **Space** structures.

A more serious problem is the space required by the **alink** field. **alink**'s are needed only in instances of ambiguously referenced objects, not in every object instance, and temporarily for building the unscanned queue. If the number of ambiguously referenced objects expected or tolerated is small, fixed-size arrays can be used.

Alternatively, at a moderate cost in space and time, the list elements can be allocated in the **old** and **new** spaces by calling **alloc**. An alist is constructed from **Vnode**s with fields **vobj**, which points to the ambiguously referenced object, and **alink**, which points to the next **Vnode** on the alist. The **limit** and **alist** fields in **Space** structures point to **Vnode**'s. As **old**'s alist is constructed, **Vnode**'s are allocated in **new** space by **aref**:

```
void aref(Object *p) {
        Vnode **q, *r, *t;

        if (inSpace(p, old) && (p->size&AOBJECT) == 0) {
                q = &old->alist;
                for (r = *q; p > r->vobj; r = *q)
                        q = &r->alink;
                t = (Vnode *) alloc(sizeof(Vnode), new);
                if (t == 0)
                        error("memory overflow during collection");
                t->vobj = p;
                t->alink = r;
                *q = t;
                p->size |= AOBJECT;
        }
}
```

Ambiguously referenced objects are marked as such because objects no longer have **alink** fields. The **AOBJECT** marks are cleared in **gc**, as shown below. **alloc** is modified accordingly by changing all occurrences of **s->limit** that refer to a ambiguously referenced object to **s->limit->vobj**. Note that if, as in many applications, **Vnode**s are small compared to objects, they may be allocated in the space between **s->next** and **s->limit** that would otherwise be lost because of fragmentation.

The unscanned queue can be threaded through the **old** space copies of non-ambiguously referenced objects if all objects have enough room for two pointers. Space for one pointer, **fields[0]**, is already required to point to the **new** space copy, and objects in many applications have space for a second pointer. Here, **fields[0]** can also be used as an element of the unscanned queue, and **fields[1]** points to the next object on the queue. With these changes, **move** becomes

```
Object *move(Object *p) {
        int i;
        Object *q;

        if (!inSpace(p, old) || (p->size&AOBJECT))
                return p;
        if ((p->size&MARKED) == 0) {
                q = (Object *) alloc(sizeInBytes(p->size), new);
                if (q == 0)
                        error("memory overflow during collection");
                q->size = p->size;
                for (i = 0; i < p->size; i++)
                        q->fields[i] = p->fields[i];
                p->fields[0] = q;
                p->fields[1] = 0;
                p->size |= MARKED;
                *tail = q;
                tail = &q->fields[1];
        }
        return p->fields[0];
}
```

These revised representations for the alist and unscanned queue lead to the revised gc:

```
gc(void) {
        int i;
        Space *t;
        Object *p;
        Vnode *q;

        old->alist = (Vnode *) old->end;
        for (i = 0; i <= sp; i++)
                aref(stack[i]);
        old->limit = old->alist;
        head = 0;
        tail = &head;
        for (q = old->alist; (char *) q->vobj != old->end; q = q->alink)
                scan(q->vobj);
        for (q = new->alist; (char *) q->vobj != new->end; q = q->alink)
                scan(q->vobj);
        while (head) {
                p = head;
                head = head->fields[1];
                if (head == 0)
                        tail = &head;
                scan(p);
        }
        for (q = old->alist; (char *)q->vobj != old->end; q = q->alink)
                q->vobj->size &= ~AOBJECT;
        t = old;
        old = new;
        new = t;
        new->next = new->start;
}
```

The last **for** loop in **gc** unmarks the ambiguously referenced objects in **old** space, which were marked by **aref**, above.

Discussion

As presented, accessible objects are reached only through ambiguously referenced objects. In most applications, there are some pointers to known objects in the root set, which may or may not be referenced by an ambiguous pointer. This situation can be handled by adding statements **p = move(p)** for each of these objects, **p**, after **old**'s alist has been constructed in **gc**. This addition has the effect of copying the known objects to **new** space and adjusting the root pointer accordingly, unless the object was also referenced by an ambiguous pointer in which case it remains in **old** space.

When there are no ambiguous references, the algorithm is equivalent the standard copying algorithm. The additional overhead in this case is the scan for ambiguous references, but this overhead must be paid by any algorithm that copes with ambiguous references. The additional space required is minimal: A few extra pointers in **Space** structures.

When there are ambiguous references, both additional time and space are required. For N ambiguously referenced objects, the insertion sort in **aref** may require $O(N^2)$ time, and requires $2bN$ bytes of additional space, where b is the size of a pointer. This additional space comes at the expense of future allocations since it is taken from **new** space. The last **for** loop in **gc** contributes an additional time overhead of $O(N)$.

Ambiguous references complicate **alloc** slightly. Allocation proceeds normally until **s->next** must be advanced over a ambiguously referenced object, which occurs N times. This additional $O(N)$ time overhead is amortized across the allocation of an entire memory space, however.

One of the algorithm's most serious drawbacks, which it shares with all such algorithms, is that it saves potentially many objects that are not really accessible. These include the ambiguously referenced objects and the objects to which they directly or indirectly refer. The number of ambiguously referenced objects can be minimized by reducing the probability that other data will incidently have pointer values or by reducing the size of the root set. On machines with large address spaces, locating the memory spaces in the high part of the address space tends to reduce the number of incidental pointer values.

The size of the root set can be reduced by using pointer maps or other information, such as stack frame layout, to identify potential pointers in the stack and global data instead of treating the contents of all pointer locations as potential pointers. This approach does, of course, require more assistance from the compiler or more hand work. A simple alternative is to try to avoid exhausting storage when the root set, i.e., the active stack, is large. This alternative can be accomplished by calling `gc` explicitly at appropriate points during execution. While there is a time penalty for this approach, in many applications, `gc` can be called when the system is otherwise idle, e.g., waiting for user input.

Another, potentially more serious, drawback of the algorithm is that it cannot handle derived pointers and objects must be self-identifying. `inSpace(p,s)` must determine—unambiguously—whether `p` points to an object in `s`. Doing so constrains the object layouts or their placement. For example, in systems like Icon [2], object layouts permit the head of an object to be identified given a potential pointer because the bit patterns that begin objects cannot occur elsewhere.

On other systems, such as Objective-C, objects must be placed so that addressing can be used to check if a pointer points to an object. For instance, suppose all pages (logical or physical) begin with objects. For pointer `p`, the head of the object at the beginning of the page into which `p` points can be located. From this object, all other objects on the page can be located thereby determining if `p` points to an object. This technique may increase the cost of `inSpace` significantly. It can, however, be used to handle derived pointers from the root set by having `inSpace` return a pointer to the head of the object.

An interesting area for further investigation is adapting the algorithm to do incremental [16] and generation-based [17] garbage collection. For example, in generation-based schemes, long-lived objects migrate to a memory space that is collected infrequently thereby reducing the cost of collection. The present algorithm might be extended with generations so that a space containing ambiguously referenced objects is used for long-lived storage. Doing so would effectively 'promote' ambiguously referenced objects to long-lived objects thereby avoiding further special treatment.

References

1. J. Cohen, Garbage Collection of Linked Data Structures, *Computing Surveys* **13**, 3 (Sep. 1981) 341–367.
2. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton Univ. Press, Princeton, NJ, 1986.
3. R. Harper, D. B. MacQueen, and R. Milner, Standard ML, Tech. Rep. ECS-LFCS-86–2, Dept. of Computer Science, Univ. of Edinburgh, Edinburgh, UK, Mar. 1986.
4. R. Pike, The Text Editor sam, *Software—Practice & Experience* **17**, 11 (Nov. 1987) 813–845.
5. G. D. Ripley, R. E. Griswold, and D. R. Hanson, Performance of Storage Management in an Implementation of SNOBOL4, *IEEE Trans. on Software Eng.* **SE-4**, 2 (Mar. 1978) 130–137.
6. *UNIX Timesharing System, Programmer's Manual, Ninth Edition, Volume 1*, AT&T Bell Laboratories, Murray Hill, NJ, Sep. 1986.
7. B. W. Kernighan and D. M. Ritchie, *The C Programming Language, Second Edition*, Prentice-Hall, Englewood Cliffs, NJ, Second Edition, 1988.
8. M. Caplinger, A Memory Allocator wiht Garbage Collection for C, *Proc. of the Winter USENIX Conf.*, Dallas, TX, Feb. 1988, 325–330.
9. H. Boehm and M. Weiser, Garbage Collection in an Uncooperative Environment, *Software—Practice & Experience* **18**, 9 (Sep. 1988) to appear.
10. R. R. Fenichel and J. C. Yochelson, A LISP Garbage-Collector for Virtual-Memory Computer Systems, *Comm. ACM* **12**, 11 (Nov. 1969) 611–612.
11. A. W. Appel, Garbage Collection Can be Faster than Stack Allocation, *Inf. Proc. Letters* **25**, 4 (June 1987) 275–279.
12. J. F. Bartlett, Compacting Garbage Collection with Ambiguous Roots, Tech. Rep. 88/2, Dec. Western Research Laboratory, Palo Alto, CA, Feb. 1988.
13. J. F. Bartlett, Compacting Garbage Collection with Ambiguous Roots, *LISP Pointers* **1**, 6 (June 1988) 3–12.

14. S. C. North and J. C. Reppy, Concurrent Garbage Collection on Stock Hardware, in *Functional Programming Languages and Computer Architecture (LNCS 274)*, G. Kahn (ed.), Springer Verlag, New York, 1987, 113–133.
15. B. J. Cox, *Object-Oriented Programming, An Evolutionary Approach*, Addison Wesley, Reading, MA, 1986.
16. H. G. Baker, List Processing in Real-Time on a Serial Computer, *Comm. ACM* **21**, 4 (Apr. 1978) 280–294.
17. H. Lieberman and C. Hewitt, A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Comm. ACM* **26**, 6 (June 1983) 419–429.