Accepted to Communications of the ACM. Not yet scheduled or copy edited.

Garbage Collection with Pointers to Individual Cells

Barak Pearlmutter

Abstract

We describe an extension to copying garbage collectors which allows them to operate in the presence of special pointers to individual cells, potentially within structures.

Introduction

In the heap model in which garbage collectors usually operate, the heap is an array of cells. Each cell contains either a non-pointer, to be ignored, or a pointer to a block of cells somewhere in the heap, called an *object*. The objects do not overlap. In addition, there are a bunch of cells not in the heap, called the *root set*. It is possible to determine from a cell whether it contains a pointer or not, and it is possible to determine from a pointer how long the object pointed to is.

The goal of a garbage collector is to preserve all structures that are accessible through a chain of dereferences starting at the root set, while reclaiming any storage not thus accessible. Such accessible storage is called *live*. In copying garbage collection, the best technique for most applications, a new heap is constructed, all live objects are copied to this new heap, and pointers are updated to reflect the new locations [2, 4].

Here we consider the addition of a new kind of value to be permitted in cells, namely pointers to single cells. These single cells may be inside objects that are referenced by the usual pointers to objects. For historical reasons, these pointers to single cells are called *locatives*. Locatives are similar to Pascal pointers, in that the only operations that can be performed on a locative are to dereference it or to alter the contents of the cell it points to.[1] In Pascal, pointers can not be constructed to arbitrary components of structures; but such pointers can be constructed in most other Algol descended languages, such as Modula or ADA.

Some lisp systems, such as ZetaLisp [8] and T [7], incorporate locatives into copying garbage collectors by having a special routine that allows a pointer to the surrounding object to be recovered from a locative. This allows locatives to exist at the expense of never reclaiming the storage occupied by an otherwise unreferenced object containing a cell pointed to by a locative. Also, requiring the ability to recover a pointer to the surrounding object from a locative constrains and complicates memory formats, as in garbage collectors for C [3].

In the next section we will develop an algorithm that reclaims the otherwise unreferenced cells of objects containing cells pointed to by locatives, thus alleviating these difficulties.

The Algorithm

The essential idea is to add an extra pass to the garbage collection process. As usual, all live objects are transported in the first pass. Then a second pass is made in which any cells referenced by locatives, but not transported by virtue of being inside live objects, are transported.

This could be accomplished by just adding the extra pass except for one complication: it might be the case that a cell pointed to by a locative contains a pointer to an object which is otherwise unreferenced. To account for this possibility, during the first pass all chains of locatives must be traversed, and if a pointer to an object is found at the end of the chain, that object must be transported.

This raises the possibility of circular chains of locatives, which would cause infinite loops. We propose three methods for breaking such loops.

[1] Unlike C pointers, locatives can not typically be incremented or decremented. This disallows a common use for pointers to the interior of objects: efficient scanning through an array. If these addressing tricks are to be used with locatives, either a pointer to the surrounding object must be kept alive while such use is possible, or there must be a special kind of "incrementable locative," perhaps constrained to point only into the interior of an array, from which the system can recover the identity of the surrounding object. One representation for such an "incrementable locative" would simply be an (object,locative) pair, which reduces to the previous solution.

The first is to track down the locative chain with two fingers, the second moving half as fast as the first, and to stop following the chain if the fast finger catches up with the slow one. The second is to paint cells in locative chains blue as they are traversed, and to stop following a chain when a blue cell is encountered. The third (suggested by a kind reviewer) is to break locative chains above some maximum length (say 4) by biting the bullet and transporting the entire surrounding object.

Each technique has its drawback. The first can take a total of $n^2/2$ accesses to track down a terminating chain of length n, since the chain will be encountered n times, and each time it must be followed all the way to the end. Similarly, a cyclic chain with n elements will consume $3n^2$ accesses, $2n^2$ by the fast finger and n^2 by the slow one.

The second technique only examines each element in any chain once, but requires storage for painting cells blue and time to check for blue cells. Including time to set and check colors, this comes to $3n$ accesses for a chain of length n. If it is not possible to put the color information into the cells themselves, the overhead of maintaining the color information can be considerable.

The third technique can fail to reclaim the storage from a dead object, which can be an arbitrarily large amount. Even if the algorithm is randomized, by making the number of locatives to be traverse before breaking the chain a random variable, it is possible to drive the probability of reclaiming a particular reclaimable object arbitrarily low by setting up enough locative chains leading into the object in question.

In the Oaklisp system [6], an instrumented garbage collector of this sort never detected a chain of more than three locatives, so the potential problem of long chains of locatives does not appear to be serious in practice. Following locative chains consumed such a miniscule fraction of the total garbage collection time that the first $O(n^2)$ technique was released in the production system, and is included in the sample code below.

Complexity Analysis

We shall measure the complexity of a garbage collection algorithm in terms of the number of memory accesses required. We shall ignore the root set and the overhead of following locative chains, as this overhead is negligible, at least in our implementation.

Without locatives, a copying garbage collector requires $5n + 1$ memory accesses for each live object of length n. This breaks down to $2n$ to copy it from old space to new space, 1 to plant a forwarding pointer, and $3n$ to scavenge it. The 3 for each cell scavenged is because it takes one to read it, one to get the forwarding information from old space, and one to write the new location.

The algorithm presented here requires $7n$ memory accesses for each live object of length n. This consists of $2n$ to copy it from old space to new space, n to plant forwarding pointers for each cell, and $4n$ for the two scavenging passes. The constant is 4 and not 6 because a single cell can not be both a pointer to an object and a locative, so the two memory accesses to find the forwarding information and update the cell can occur in only one of the two scavenging passes.

To summarize, allowing locatives increases the constant factor from 5 to 7.

Extensions and Applications

The many extensions to copying garbage collection, such as generation scavenging, online ephemeral garbage collection, virtual memory based techniques, et cetera [1], are all compatible with the technique presented above.

One interesting application would be to ML. It would be possible to put "`ref` α" objects that occur as substructures inline, representing the "`ref` α" itself as a locative to the involved cell, at the expense of requiring the compiler to know about two methods for retrieving structure slots: a simple fetch for non-`ref` slots, and an address computation for `ref` slots. This would save one, or in most implementations two, cells of storage per mutable variable or mutable structure slot. In our implementation of Oaklisp the major application of locatives was essentially the same, representing mutable and indefinite lifetime variables, and we found them surprisingly useful. Similarly, ZetaLisp uses locatives internally in the implementation of environments, as do many implementations of languages with first class functions.

Garbage collection with locatives is a special case of garbage collection of potentially nested structures. The ability to garbage collect generally nested structures would open the possibility of representing substructures in garbage collected languages with nested structures in memory, both speeding access and saving memory.

Although mark and sweep or in place compaction algorithms can be used with generally nested structures, nesting has been eschewed in garbage collectible structures in order to permit the use of more efficient copying collectors.

The algorithm presented here can be extended to the case of more generally nested structures by first using pointer reversal to mark all accessible cells in place, and then maintaining contiguity during the transportation phase by transporting the entire surrounding block of marked cells when any cell is transported. Even with the scan to find the surrounding block of marked cells, the number of memory accesses remains proportional to the number of live cells.

Acknowledgments

Oaklisp was implemented jointly with Kevin Lang, without whose constant encouragement and unflagging enthusiasm this this paper could not have been written. This work was performed in the Carnegie Mellon University Department of Computer Science, where the author was supported by a Hertz fellowship.

References

[1] APPEL, A. Garbage collection. In Lee [5], pp. 89–100.

[2] CHENEY, C. J. A nonrecursive list compacting algorithm. *Commun. ACM 13*, 11 (1970), 677–678.

[3] DETLEFS, D. L. Concurrent garbage collection for C++. In Lee [5], pp. 101–134. Also see thesis of same title, Carnegie Mellon University School of Computer Science technical report CMU-CS-90-119.

[4] FENICHEL, R. R., AND YOCHELSON, J. C. A lisp garbage collector for virtual memory computer systems. *Commun. ACM 12*, 11 (Nov. 1969).

[5] LEE, P., Ed. *Topics in Advanced Language Implementation*. MIT Press, 1991.

[6] PEARLMUTTER, B. A., AND LANG, K. J. The implementation of Oaklisp. In Lee [5], pp. 189–215.

[7] REES, J. A., ET AL. *The T Manual*, fourth ed. Yale University Computer Science Department, 1984.

[8] WEINREB, D. L., AND MOON, D. A. *Lisp Machine Manual*, fourth ed., July 1981.

Appendix: Sample Code

In the code below, the semantics of most of the low level routines used should be clear from their names. The calls `slot(pointer,i)` and `contents(locative)`, for getting slot `i` of the object pointed to by `pointer` and the contents of the cell pointed to by `locative`, respectively, are used as lvalues and presumably defined as macros.

```
 /* Take a pointer to an object in old space and transports the
   object, seting up forwarding pointers and returning a pointer
   to the new location. */

cell transport(cell c)
{
  int i, length = object_length(c);

  /* Allocate space and make pointer to new location. */
  cell newc = cvt_to_pointer(&new[freepoint]);
  freepoint += length;

  /* Copy the contents over and set up forwarding pointers. */
  for(i=0; i<length; i++) {
    elt(newc,i) = elt(c,i);
    elt(c,i) = locative_to(&elt(newc,i));
  }

  /* Return the new pointer. */
  return newc;
}


/* This works by explicitly detecting circular chains by keeping
   a trailing pointer that travels at half the speed of the
   leading one.  If the leader ever catches the trailer, a
   circularity has been detected. */

void follow_loc_chain(cell c)
{
  cell slow = c;
  bool advance_slow = FALSE;

  while (old_locative(c)) {
    c = contents(c);
    if (c == slow) return;
    if (advance_slow) slow = contents(slow);
    advance_slow = !advance_slow;
  }
  (void)touch(c);
}


/* Return the new version of c, transporting objects if
   necessary.  Locatives to cells in old space are not
   transported by this routine. */

cell touch(cell c)
{
  if (old_pointer(c))
    if (new_locative(elt(c,0)))
      return cvt_locative_to_pointer(elt(c,0));
    else return transport(c);
  else {
    if (old_locative(c)) follow_loc_chain(c);
    return c;
  }
}


/* Passed a locative to a cell to be transported. */

cell loc_transport(cell c)
{
  cell newc = locative_to(&new[freepoint++]);
  contents(newc) = contents(c);         /* transport */
  contents(c) = newc;                 /* forward */
  return newc;
}


/* Return the new version of c, no matter what c is.  If c is a
   locative to a cell in old space, that cell is moved to new
   space if necessary.  If c is a pointer to an object in old
   space, that object should already be transported. */
```

```
cell loc_touch(cell c)
{
  if (old_pointer(c)) return cvt_locative_to_pointer(elt(c,0));
  else if (!old_locative(c)) return c;
  else if (new_locative(contents(c))) return contents(c);
  else return loc_transport(c);
}


/* Main routine. */

void gc()
{
  int i;

  /* Nothing in new space yet: */
  freepoint = 0;

  /* Transport objects pointed to by the root set. */
  for (i=0; i<ROOT_SIZE; i++)
    root[i] = touch(root[i]);

  /* Scavenge new space for pointers to objects. */
  for (i=0; i<freepoint; i++)
    new[i] = touch(new[i]);

  /* Transport any solitary cells pointed to by the root set. */
  for (i=0; i<ROOT_SIZE; i++)
    root[i] = loc_touch(root[i]);

  /* Scavenge new space for locatives to solitary cells. */
  for (i=0; i<freepoint; i++)
    new[i] = loc_touch(new[i]);
}
```

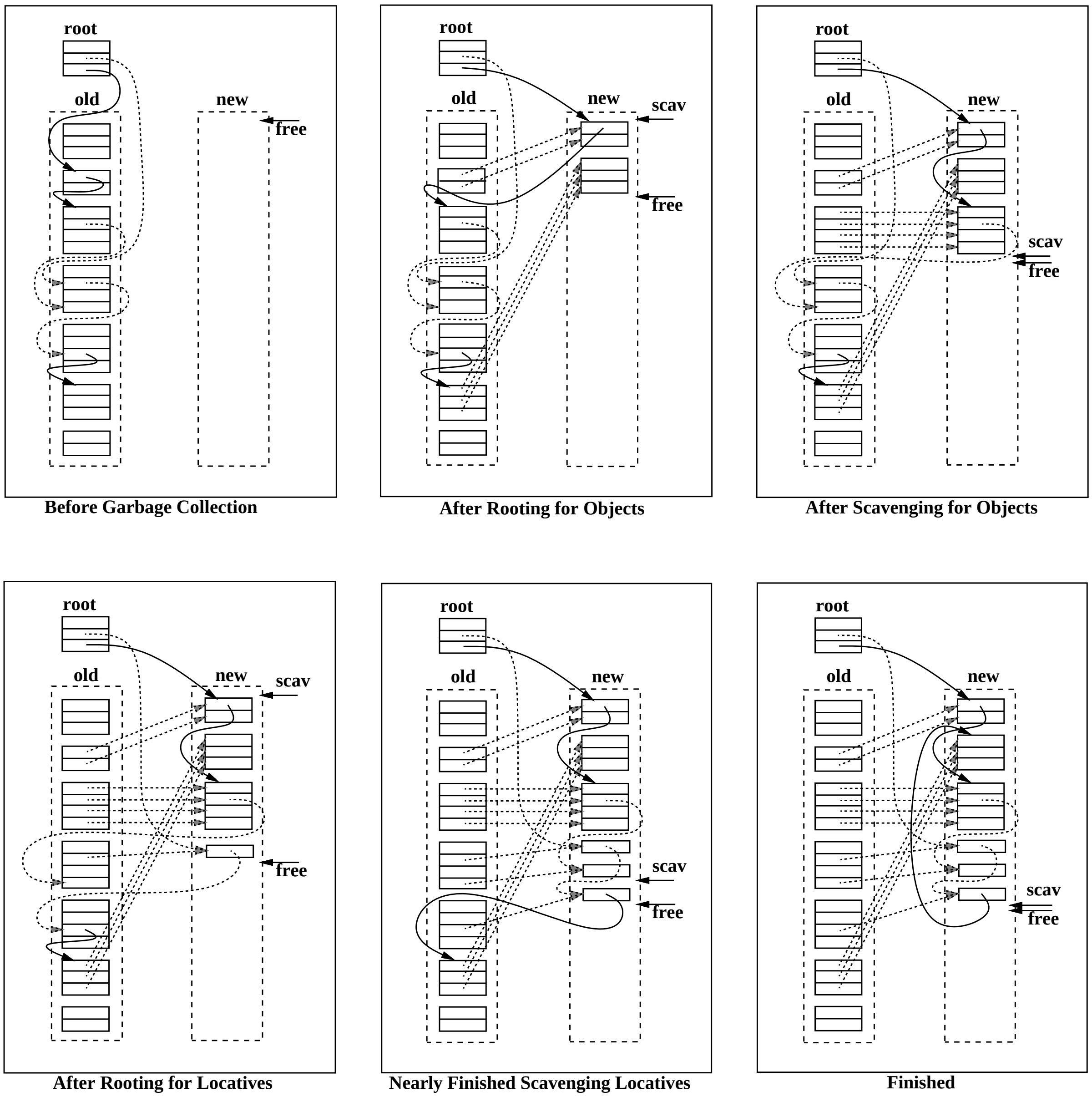

Figure 1: The garbage collector at work. Solid lines are pointers to objects and dotted ones are pointers to individual cells.