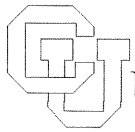
Imminent Garbage Collection

John Shultis

CU-CS-305-85



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

IMMINENT GARBAGE COLLECTION

by

Jon Shultis

CU-CS-305-85 July, 1985

University of Colorado, Department of Computer Science,
Boulder, Colorado.

Imminent Garbage Collection

Jon Shultis
Department of Computer Science
University of Colorado
Boulder, CO 80309

Abstract

Imminent garbage collection is a new dynamic storage optimization technique. Imminent garbage is storage which has been allocated and is accessible, but for which it can be determined that no future access will be made. We present an efficient algorithm for detecting and reclaiming imminent garbage using a constant amount of storage overhead per allocated storage structure.

Imminent Garbage Collection

Jon Shultis
Department of Computer Science
University of Colorado
Boulder, CO 80309

Garbage is storage which has been allocated during the execution of a program but which is no longer accessible [3]. *Imminent garbage* is storage which has been allocated and is accessible, but which will no longer be accessed. A common source of imminent garbage is the sequence of activation records of a tail-recursive procedure up to, but not including, the most recent activation.

Imminent garbage collection is the process of detecting imminent garbage and making it available for re-use. Although the creation of imminent garbage can sometimes be avoided by static program transformation (for example, by transforming tail recursion to iteration [1]), this is not always possible, because the point at which a block of dynamically allocated storage is no longer needed depends on its dynamic context. Hence imminent garbage collection makes it possible for some programs to complete which would otherwise be terminated for lack of storage.

As a simple example, consider the (somewhat artificial) recursive search procedure below.

```
function search(node)
begin
  if node↑.key < target then return search(node↑.left) + 1
  elsif node↑.key = target then return node↑.data
  else return search(node↑.right)
  end if;
end search;
```

When node↑.key > target, the current activation of search makes a "tail recursive" call, after which the current activation is no longer needed - it is imminent garbage.

Programs that use large numbers of small procedures generally produce more imminent garbage than those with a few large procedures, because of the smaller granularity of allocated storage; it is simply more probable that one or two items can be discarded at any time than that a large number can.

For example, suppose that, in the procedure search, the expression search(node↑.left)+1 were replaced by succ(search(node↑.left)). As soon as succ is entered, the invoking activation of search becomes imminent garbage, whereas in the original program the incrementing of search(node↑.left) was performed in-line, requiring the original activation of search to be retained. Although no real benefit is realized in this case, one need only imagine in place of search a procedure having many parameters and local storage amounting to several hundred bytes to see the advantage of retaining the activation record for plus instead of that for search. Programs that make heavy use of recursion and higher-order functions are especially good generators of imminent garbage.

We describe an algorithm for imminent garbage collection in statically scoped languages. For purposes of discussion, we assume that programs are compiled, and that all non-local branches (including jumps into and out of blocks, label variables, procedure values, and procedure returns) are represented by explicit closures consisting of an instruction pointer (ip) and an environment pointer (ep). Since we do not wish to limit our discussion unnecessarily, we assume that activation records are allocated in a heap, with an explicit free storage list. Our algorithm is easily adjusted for use with a stack by adding a compaction stage to close up the gaps in the stack created by the release of imminent garbage.

With these assumptions, the unit of storage allocation is the activation record; we do not consider individual variables. Hence the problem of detecting

imminent garbage is, for us, the problem of detecting entire activation records that are no longer needed.

Incidentally, if the language allows the programmer to manipulate explicit references to objects in an auxiliary heap, the management of that heap is irrelevant to our discussion. When we recycle an activation record, explicit references simply become inaccessible sooner than they would have otherwise. In this case, imminent garbage collection simply has the nice side effect of making explicit heap storage available for (independent) reclamation early, in addition to freeing implicit (activation record) storage.

There are two main steps in the detection of imminent garbage. The first step is to determine when an activation record is in a state of *imminent return.* Intuitively, we need to know when all future execution in the environment defined by an activation record leads to exiting that environment without referencing any variables (local or global). The second step is to determine when an activation record is *obscured.* Intuitively, an activation record is obscured if its storage will never be accessed from any other activation. Taken together, imminent return and obscurity define imminent garbage.

1. Imminent Return

In order to define the notion of "imminent return" formally, we need two subsidiary definitions.

Let C be the set of all extant closures $<ip,ep>$. The set R_α of *resumptions* of an activation record α is the set of instruction pointers corresponding to the continuation points of procedures executing in the environment defined by α. Formally,

$R_\alpha \triangleq \{ip \mid <ip,\alpha> \in C\} \cup \{pc \mid arbase = \alpha\}$,

where *pc* is the current value of the program counter, and *arbase* is the current activation record base register. Notice that, in order to compute R_α, we must be able to compute C, which is generally infeasible if the language allows the programmer to store closures in untagged unions. Similarly, it may not be possible to identify references as such if an *address* operation is provided, or if references can be converted to other data types, as in C. For such languages, imminent garbage collection is not possible.

Given an instruction pointer *ip*, we define *nso*(*ip*) to be the *next substantive operation starting from ip*. Basically, *nso*(*ip*) is found by following any unconditional branches, ignoring *skip* instructions, and so forth.

With these definitions, an activation record α is in a state of *imminent return* if

$\{op \mid op = nso(ip) \wedge ip \in R_\alpha\} = \{return\_from\_subroutine\}$

In other words, the only significant action that can ever be taken in α at any future time is to return to a caller.

2. Obscurity

If an activation record α is in a state of imminent return, then no resumption of α depends on α. If we can determine that no resumption of any other activation record will ever access α as global data, then α is effectively waiting around for the control thread to sever it from the rest of the heap; it is imminent garbage. An activation record that will never be accessed globally is said to be *obscured*. In order to define obscurity formally we again need some auxiliary definitions.

The *globals* of α, denoted by G_α, is the set of activation records which are accessible from α via the static chain. Formally, let $sl(\alpha)$ denote the static link of

α. Then, given that α's static level is $k \geq 0$,

$G_\alpha \triangleq \{sl^i(\alpha) \mid 1 \leq i \leq k\}$

where sl^i denotes the i-fold composition of sl with itself.

G_α tells us which activation records are accessible from α; we want to know, however, from which activation records a given activation record is accessible. To do this, we simply invert G_α to get E_α, the *entourage* of α. Formally,

$E_\alpha \triangleq \{\beta \mid \alpha \in G_\beta\}$

E_α specifies the set of activation records from which α could in principle be accessed. This set, ordered by static level, forms a lower semilattice with α as least element. In particular, note that there may be several activation records at relative static level j in α's entourage.

We get a crude measure of obscurity if we equate "could be accessed" with "could be accessed in principle": α is obscure if all of its entourage is in a state of imminent return. A more general definition of this concept follows.

For each procedure in a program, we can determine statically a point after which we can guarantee that no further references will be made to global variables at relative static levels $\geq j$; call this the set of *j-points* of the procedure. A crude determination of the j-points is to use the end of the procedure for all j, as we did above. A more refined determination can be made by live variable analysis [2].

The table of j-points must be computed at compile-time, and retained for use by the imminent garbage collector at run-time. For a program consisting of p procedures with a maximum static nesting level of n, the j-point table requires $O(pn)$ storage, which is tolerable overhead.

Multiple procedure exits do not pose a serious problem. In place of a single point for each j we have a set of points, one for each exit. For each exit, the

corresponding j-point marks the beginning of a basic block ending at that exit. The interval between the beginning of this block and the corresponding exit is called a j-block. In what follows, we then say that an instruction pointer *ip* is past its j-point if it lies within some j-block.

Define the *j-circle* of α to be that subset of its entourage for which α is at relative static level j. We can guarantee that α will not be referenced from its entourage if the next substantive operations of all resumptions of all members of α's j-circle are either subroutine returns or are past the j-point of the procedure. When this occurs, we say that α is *obscured.*

Finally, we define α to be *imminent garbage* if it is obscured and in a state of imminent return. Obscurity guarantees that the local storage of α is not needed by any resumption of any other activation record. It also guarantees that α's static link is no longer needed; that is why we include all levels $\geq j$ in determining the j-points. α's imminent return guarantees that no resumption of α itself requires either the local storage of α, or its static link. The only part of α that remains is its return closure (that is, the return address and dynamic link).

If α is in a state of imminent return, then all resumptions *ia* of α have the effect of setting the program counter to *ia* and activation base to α, executing some instructions that have no effect on the outcome of the program, and then setting *pc* and *arbase* to the values in α's return closure. So, all we need to do to eliminate the remaining dependence on α is to replace each closure containing a resumption of α by α's return closure!

3. A Practical Algorithm

Since we are interested in collecting imminent garbage when available storage is exhausted, a practical algorithm should require either no additional storage, a

fixed amount of storage, or a fixed amount of storage per activation record. The algorithm we present here uses three bits plus two small natural numbers in each activation record, one bit plus two small natural numbers in each object allocated in the explicit heap, if one exists, and a fixed table of size $O(pn + t)$, where p is the number of procedures in the program, n is the maximum nesting level, and t is the number of data types for which explicit heap storage is required. The algorithm detects imminent garbage in one traversal of the heap, and recycles it in a second traversal. During the recycle pass the heap is prepared for the next detection pass, so there is no explicit initialization pass. The detection pass takes $O(inc + ec)$ time, where i is the number of activation records (objects in the implicit heap), e is the number of objects in the explicit heap, and c is the average number of closures stored in an object. The recycle pass takes $O(i+e)$ time. If we discount c and n as being fixed small numbers for any given program, the overall time performance is $O(i+e)$, i.e. linear in the number of allocated objects.

At compile time, we construct a table of *templates*, one for each procedure and one for each type of object for which explicit heap storage is required. A template is simply the collection of data needed to navigate through an activation record or other program object to find closures and references to other objects that may contain closures.

Procedure templates also include a vector of j-points for that procedure, indexed by relative static level starting at 0. So, for example, any resumption of a procedure at an address greater than or equal to the third entry in the procedure's j-point vector is guaranteed to reference global variables at a relative static level of at most three. Since the details of templates depend heavily on the specific programming language, we do not describe them further here.

An *imminent return indicator* bit, an *obscured* bit, and a *processed* bit are included in each activation record to guide the traversal. The number of the corresponding template is also stored in each activation record, along with the current processing *position* in the template. The *processed* bit, template number, and position are also kept in each explicit heap object.

Assume that all indicator bits are initially set. The imminent return indicator of an activation record is cleared if a resumption of that record is found for which the next substantive operation is not a return_from_subroutine (fig. 1). The *obscured* bit of an activation record is cleared if some resumption may make references to global variables in that activation record (fig. 2).

An adaptation of the Schorr-Waite algorithm [4] is used for traversing the heap (fig. 3). The traversal requires three auxiliary pointer variables, named *current, parent,* and *temp.* The variable *current* points to the structure being processed, and *parent* refers to the structure where processing should be resumed after

```
procedure imminent_return(ip,ep);
begin
if (nso(ip) ≠ return_from_subroutine) then
  clear(ep↑.imminent_return_indicator);
end imminent_return;
```

figure 1

```
procedure obscured(ip,ep);
var no_levels_needed:0..n;
begin
no_levels_needed := min { k | ip≥templates[ep↑.templateno].j_points[k] };
while no_levels_needed>0 loop
  ep := ep↑.static_link;
  clear(ep↑.obscured);
  no_levels_needed := no_levels_needed - 1;
end loop;
end obscured;
```

figure 2

```
current := make_root(pc,arbase);
parent := nil;
current↑.position := 1;

detect: loop
  (ip,ep) := closure_at(base => current,
             offset => templates[current↑.template_no].closures[current↑.position]);
  if nil_closure(ip,ep) then -- resume processing parent
    exit when parent=nil;  -- end of detection pass
    (ip,ep) := closure_at(base => parent,
               offset => templates[parent↑.template_no].closures[parent↑.position]);
    cycle_up;
    increment current↑.position;
  else
    imminent_return(ip,ep);
    obscured(ip,ep);
    if ep↑.processed is set then
      clear(ep↑.processed);
      cycle_down;
      current↑.position := 1;
    else
      increment current↑.position;
    end if;
  end if;
end loop detect;
```

figure 3

the processing of *current* is complete. The position in the parent structure's template where processing should resume is represented by a small natural number (essentially an offset in the template), and is stored in *parent↑.position.*

Each closure <*ip,ep*> in *current↑* is examined in turn, using the offsets listed in *templates*[*current↑.template_no*] as a guide. *imminent_return*(*ip,ep*) and *obscure*(*ip,ep*) are computed, and then *ep↑*'s *processed* bit is checked. If it is set, it is cleared and the structure at *ep* is processed, otherwise processing proceeds to the next closure by incrementing *current↑.position.*

To begin processing *ep*, *ep* is replaced by *parent*, *parent* is replaced by *current*, and *current* is replaced by *ep*, using *temp* as an intermediary in the exchange (fig. 4a). When the processing of *current* is complete, the original configuration of these

pointers is recovered by *cycle_up* (fig. 4b). After the parent has been thus restored as the *current* structure, processing proceeds to the next closure by incrementing *current↑.position*. The processing of references to structures in the explicit heap is essentially the same, treating the reference as though it were the environment pointer of a closure.

When the detection traversal is complete, all structures involved in the traversal have had their *processed* bits cleared. Any activation record having a set *imminent_return_indicator* or *obscured* bit is, in fact, in a state of imminent return or obscured, respectively, since no evidence to the contrary was unearthed during the traversal.

Imminent garbage is moved to the free list during the collection traversal (fig. 5). Those activation records for which both the *imminent_return_indicator* and *obscured* bits are set are linked onto the free list, using the static link field (since their return closures must be preserved until the end of the traversal). Each clo-

```
procedure cycle_down;
begin
temp := current;
current := ep;
ep := parent;
parent := temp
end; -- cycle_down
```

figure 4a

```
procedure cycle_up;
begin
temp := parent;
parent := ep;
ep := current;
current := temp
end; -- cycle_up
```

figure 4b

sure *<ip.ep>* such that both *ep↑.imminent_return_indicator* and *ep↑.obscured* are set is replaced by *ep↑.return_closure.*

The *processed* bits, which were cleared during detection, are reset during collection, so that they are all set at the beginning of each cycle. However, if the *imminent_return_indicator* and *obscured* bits are reset during collection, then some activation records will be mistaken for imminent garbage. Instead, these bits are cleared unless they are both set (i.e., they are left intact for imminent garbage, and cleared otherwise). The subsequent cycle must then reverse the sense of these bits. The modifications needed to make *imminent_return* and *obscured* set and clear

```
current↑.position := 1;
collect: loop
  (ip,ep) := closure_at(base => current,
             offset => templates[current↑.template_no].closures[current↑.position]);
  if nil_closure(ip,ep) then -- resume collecting parent
    exit collect when parent = nil; -- end collect pass
    (ip,ep) := closure_at(base => parent,
               offset => templates[parent↑.template_no].closures[parent↑.position]);
    cycle_up;
    increment current↑.position;
  elsif ep↑.imminent_return_indicator is set
      ∧ ep↑.obscured is set then -- ep↑ is imminent garbage
    if ep↑.processed is cleared then  -- first encounter
      link_to_freelist(ep);
      set(ep↑.processed);
    end if;
    closure_at(base => current,
          offset => templates[current↑.template_no].closures[current↑.position]) :=
          return_closure(ep↑);
  elsif ep↑.processed is set then
    increment current↑.position;
  else
    cycle_down;
    set(current↑.processed);
    clear(current↑.imminent_return_indicator);
    clear(current↑.obscured);
    current↑.position := 1;
  end if;
end loop collect;
```

figure 5

these bits on alternate cycles are straightforward exercises.

4. A Better Algorithm

There is a major flaw in the algorithm presented above. The closures contained in imminent garbage have been allowed to affect the decision about whether or not other structures are obscured, when in fact those closures will never be used. A second collection cycle may well discover more imminent garbage, and a third still more, etc. Eventually, of course, things must stabilize, but we would prefer to find as much imminent garbage as possible in a single cycle.

As it happens, the flaw originates in our original definition of R_α, the resumptions of α, which considered *all* closures having α as environment pointer. If we want to detect as much imminent garbage as possible, then we must limit consideration to those closures that are not contained either in imminent garbage, or in explicit heap structures that are accessible only through imminent garbage. Of course, restricting the set of resumptions to just these *valid* resumptions makes the entire set of definitions recursive. The ultimate affect on our algorithm would be to have it make repeated traversals to achieve a fixed point!

There is, however, a better solution. Recall that, during collection, the return closures of imminent garbage replace the closures that refer to that imminent garbage. When processing a closure $<ip,ep>$ in the detection pass, if $ep\uparrow$ appears to be imminent garbage, then we (effectively) replace $<ip,ep>$ by $ep\uparrow.return\_closure$, anticipating the replacement that will occur if $ep\uparrow$ does turn out to be imminent garbage.

In practice, no actual replacement is done. Instead, we arrange the templates so that the return closure field is the next-to-last entry, and the static link is last. (The reason for putting the static link last will be explained presently.) $ep\uparrow$ is then

processed starting at the return closure field, but its *processed* bit is not cleared, since it hasn't been completely processed.

A potential problem with this approach is that the only opportunity in the algorithm for processing a structure is when a closure referring to that structure is encountered. If a structure is passed over because it appears to be imminent garbage, but its *obscured* bit is later cleared, there may be no other closure referring to that structure, in which case it will never be processed, when it clearly should be.

To ensure that all structures that must be retained are processed, the activation record pointed to by the static link of *current*↑ is processed whenever that activation record's *obscured* bit is found to be cleared (fig. 6). This happens regardless of whether or not that activation record is marked as having been processed already, because an invocation of *obscured* may mark the static chain leading from an activation record to a greater depth than was marked when that record was originally processed. The amount of redundant processing entailed by chasing static links is reduced by processing only the static link of any record that has been previously processed. To make it as easy as possible to process only the static link field, we assume that this is the last item in the corresponding template, so that the end of the template is encountered immediately after processing of the static link is complete.

Because the static link is not a closure, the *ip* returned by closure_at for this field is *nil*. The *nil ip* signals that the computation of imminent_return and obscured should be bypassed, and that ept's static link should be processed even if ept's *processed* bit is cleared.

```
current := make_root(pc,arbase);
parent := nil;
current↑.position := 1;

detect: loop
  (ip,ep) := closure_at(base => current,
             offset => templates[current↑.template_no].closures[current↑.position]);
  if nil_closure(ip,ep) then -- resume processing parent
    exit when parent=nil;  -- end of detection pass
    (ip,ep) := closure_at(base => parent,
               offset => templates[parent↑.template_no].closures[parent↑.position]);
    cycle_up;
    increment current↑.position;
  elsif ip=nil then -- processing static link field
    if ep↑.obscured is cleared then -- chase static link
      cycle_down;
      if current↑.processed is set then
        clear(ep↑.processed);
        current↑.position := 1;
      else
        current↑.position := static_link_field; -- last entry in template
      end if;
    else -- don't chase static link
      increment current↑.position;
    end if;
  else -- processing a closure field
    imminent_return(ip,ep);
    obscured(ip,ep);
    cycle_down;
    if current↑.processed is set then
      if current↑.imminent_return_indicator is set
        ∧ current↑.obscured is set then -- looks like garbage to me!
        current↑.position := return_closure_field; -- next to last entry in template
      else -- process the whole record
        clear(current↑.processed);
        current↑.position := 1;
      end if;
    else -- process only the static link field
      current↑.position := static_link_field;
    end if;
  end if;
end loop detect;
```

figure 6

By allowing records to be processed more than once, have we introduced the possibility of nontermination? No. Only the static links of previously processed records are considered, and these form a lower semilattice, so it is not possible to

traverse a cycle in the heap by following these links. In short, our suggestion that we process only the static links to reduce the amount of redundant processing is a gross understatement!

The modified detection algorithm requires a small modification to the collection algorithm, because the *processed* bits of imminent garbage are never cleared. All that is required is to clear them when they are first encountered and linked onto the free list, instead of setting them. In subsequent cycles, the sense of these bits must then be reversed, as with the *obscured* and *imminent_return_indicator* bits.

Acknowledgements

I thank Paul Harter for his thoughtful comments on an early draft of this paper.

References

1. J. Darlington and R. M. Burstall, A System Which Automatically Improves Programs, *Acta Informatica 6*, (1976), 41-60.

2. M. S. Hecht, *Flow Analysis of Computer Programs*, North Holland, New York, 1977.

3. D. E. Knuth, *Fundamental Algorithms, 2nd Edition*, Addison-Wesley, 1973.

4. H. Schorr and W. M. Waite, An Efficient Machine-independent Procedure for Garbage Collection in Various List Structures, *Comm. ACM 10*, 8 (Aug. 1967), 501-506.