Guardians in a Generation-Based Garbage Collector

R. Kent Dybvig, Carl Bruggeman, and David Eby

Indiana University
Computer Science Department
Lindley Hall 215
Bloomington, Indiana 47405
{*dyb,bruggema,deby*}*@cs.indiana.edu*

Abstract

This paper describes a new language feature that allows dynamically allocated objects to be saved from deallocation by an automatic storage management system so that clean-up or other actions can be performed using the data stored within the objects. The program has full control over the timing of clean-up actions, which eliminates several potential problems and often eliminates the need for critical sections in code that interacts with clean-up actions. Our implementation is "generation-friendly" in the sense that the additional overhead within a generation-based garbage collector is proportional to the work already done there, and the overhead within the mutator is proportional to the number of clean-up actions actually performed.

1 Introduction

Many programming systems, such as Scheme, Common Lisp, ML, and Prolog, support dynamic allocation and automatic deallocation of objects. Within these systems, some form of storage manager takes responsibility for handling both requests to allocate new objects and automatic deallocation of objects that are no longer needed. Most current automatic storage managers employ a garbage collector [8] to deallocate unneeded objects, although some also employ reference counting mechanisms[1] [2].

Automatic storage management is useful for several reasons. First, it simplifies the programmer's job, eliminating the burden of freeing dynamically-allocated objects and the need to decide which part of a possibly complex system is responsible for freeing shared storage. Second, it can be more efficient than explicit freeing of objects. Modern garbage collectors run in time proportional to the amount of data retained in the system rather than the amount freed; in most cases, this results in far less overhead than explicit freeing, which is proportional to the amount of data freed. Third, it eliminates the possibility of storage leaks. Finally, most importantly, it eliminates the possibility of dangling references, *i.e.*, references to deallocated storage. This is especially important in systems that purport to guarantee the type safety of all memory references.

The collector considers an object to be unneeded when no pointers to the object remain, *i.e.*, when the object is no longer accessible from within the rest of the program, conventionally called the *mutator*. In most cases, this is ideal; if an object is inaccessible, how could any part of the mutator still need the storage associated with it, or more particularly, the information contained within that storage? In fact, however, there are cases in which the object or the information contained within the object may still be needed or may at least be useful. Often, these cases concern the need to *clean up* or deallocate some resource (perhaps external) associated with the object.

For example, files in Scheme are represented by *ports*. Ports encapsulate a file identifier, used to perform operating system requests for primitive I/O operations, a buffer containing unread or unwritten data, and various other items of information relating to the file or buffer. Because of exceptions and nonlocal exits, a port may not be closed explicitly by a user program before the last reference to it is dropped. This can tie up system resources and may result in data associated with output ports remaining unwritten until the system exits. It is important, therefore, for a Scheme implementation to arrange to flush unwritten data and close a port when the port becomes inaccessible.

Scheme programs that employ external library routines must often cope with external resources required

[1] Although the ideas presented in this paper can be extended to reference counting systems, we limit our discussion to collector-based systems.

Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation, pp. 207–216.

by those routines, in particular, with external memory managed with the Unix `malloc` and `free` procedures or their equivalent. In order to simplify deallocation of external memory, a Scheme header can be created for each block of storage, and a clean-up action associated with the Scheme header could then be used to free the storage. Similar mechanisms can be used to free other external resources, such as temporary files and subprocesses.

A different but related problem arises in the management of hash tables. Hash tables provide a convenient and efficient way of attaching values to a set of *keys*, where each key is an arbitrary Scheme object. Hash tables can be used to represent symbol tables and to support shared structure detection during the printing of directed acyclic and cyclic graph structures. While a hash table must retain both the key and value as long as some possibility exists for the value to be accessed through the key in the table, the key/value association should be dropped from the table whenever the key becomes inaccessible outside of the table. *Weak pairs*, which are discussed in the following section, can be used to construct the hash table in such a way that the keys are dropped automatically by the collector, but they do not support removal of the values associated with dropped keys without a periodic scan of the entire table.

Sometimes it is useful to maintain an internal "free list" of objects that are expensive to allocate or initialize. Support for automatically returning such objects to the free list when they would otherwise be reclaimed can lead to a simpler, more efficient, and more robust implementation. This might be true, for example, of a set of large objects (such as a set of bit maps representing graphical displays) whose structure and/or contents remain fixed once they are initialized. In order to save the cost of rebuilding or reinitializing new storage locations, it may be less time consuming to reuse a freed object if one exists.

These problems all have in common the need for an object to be saved from destruction once it becomes inaccessible so that *finalization*[2] actions involving the object can be performed. Essentially, we would like to extend the benefits derived from automatic storage management, as described above, to external resources and higher level internal storage management.

There are four important issues to consider in the design of a finalization mechanism:

- When does finalization occur? If finalization is done by the collector, then all access to structures shared by the mutator and finalization routines must be done within a critical section because the collector may interrupt the mutator at any point. For example, if the mutator is updating a hash table when a garbage collection occurs (with the table in an inconsistent state), restructuring the hash table at the end of the collection to remove unneeded bindings would corrupt the table.

- In what order are objects finalized? For cyclic or shared structures it may be important to finalize related objects in a particular order.

- Is the full range of language features available to finalization routines? For example, can allocation be done? Can another collection occur? What happens if a a finalization routine signals an error?

- Is the object being finalized available to the finalization routine? If so, can it be let loose into the system again? Can objects being finalized be re-registered for finalization?

There is a fifth issue to consider for generation-based garbage collectors, which segregate objects based on their ages and scan older objects less frequently than newer objects [9]. A "generation-friendly" finalization mechanism must insure that the overhead for finalization is (at worst) proportional to the amount of work already done by the collector. Among other things, this means that there should be no additional overhead for older objects that are not being collected during a particular collection cycle. Furthermore, overhead in the mutator should be proportional to the number of objects for which clean-up actions are actually performed; it does no good to eliminate the overhead of scanning older objects in the collector if the mutator must do so. In particular, scanning through an entire hash table, as described above, in order to eliminate the values for keys that have disappeared is unacceptable.

The ideal answers to the first two questions above, "when does finalization occur" and "in what order", depend on the particular application. This led us to design a mechanism that gives the program complete control over when and in what order finalization occurs. The *guardian* mechanism that resulted also permits unrestricted access to all language features and makes the object available to the finalization routine without restrictions.

Guardians provide a means to protect objects from destruction by the garbage collector. A guardian is an object with which objects can be registered for preservation and from which objects actually saved from destruction can be retrieved, one at a time, at the convenience of the program. New guardians are created dynamically using *make-guardian*, the single new primitive required by this mechanism. An object may be registered with more than one guardian or registered multiple times with a guardian. Finalization of a group

[2]Following traditional usage, we use the term "finalization" even though the actions may not really be "final."

of objects can be canceled by simply dropping all references to the guardian.

We have added guardians to our generation-based collector. The cost for handling guardians in both the collector and mutator is very small, and there is no overhead for older objects except when they are subject to collection. The collector also supports weak pairs (mentioned above and described in the following section), which complement the guardian mechanism.

The remainder of this paper is organized as follows: Section 2 discusses existing mechanisms. Section 3 describes the guardian mechanism and gives examples of its use. Section 4 discusses the implementation of guardians and weak pairs within the framework of a generation-based garbage collector. Section 5 summarizes the paper and presents a somewhat more general interface to the same basic mechanism.

2 Background

Guardians are related to the *weak sets*[3] provided by the T language [11]. A weak set is a data structure containing a set of objects. Operations are provided to add new objects, remove objects, and retrieve a list of the objects in the set. Weak sets are so-called since they maintain "weak" pointers to the objects in their sets. A weak pointer to an object is treated like a normal pointer by the garbage collector as long as nonweak pointers to the object exist. If only weak pointers to an object exist, however, the pointers are "broken" and the object is released. As a result, an object that is not accessible except by way of one or more weak sets is ultimately discarded and removed from the weak sets to which it belonged.

MultiScheme [10] provides a similar, more primitive feature, *weak pairs*. Weak pairs are like normal pairs except that the "car" (data) field of the pair is a weak pointer. The "cdr" (link) field is a normal pointer.

MIT Scheme and recent versions of T support a "weak hashing" feature that provides a form of weak pointer. The primitive *hash* accepts an object and returns an integer that is unique to that object, *i.e.*, the same integer is never returned for a different object. The primitive *unhash* accepts an integer and returns the associated object, if the object has not be reclaimed by the garbage collector. If the object has been reclaimed, *unhash* returns false. The integer can be used as a weak pointer to the object.

Weak sets, weak pointers, and weak hashing are essentially equivalent for our purposes; each allows the program to maintain a pointer to an object that is ultimately broken once the object becomes otherwise inaccessible. Any data contained within the object, however, is lost when the pointer is broken. In spite of this, weak pointers can be employed to solve the sorts of problems described in the preceding section if we are willing to introduce an extra level of indirection. Instead of maintaining a pointer directly to the data, the program can maintain a weak pointer to an object header containing a nonweak pointer to the data. If a separate nonweak pointer to the data is maintained, then when the weak pointer to the header is broken the data needed to perform the clean-up action is still available.

There are several problems with this solution. The extra level of indirection causes additional complexity since any part of a program that might receive the object must know about the indirection and adjust for it. For this reason, it is inherently unsafe, since it is possible for some part of a program to keep a pointer to the data itself even after the header has been dropped. This problem is addressed by Atkins [1], who proposes the use of a *forwarding object* that causes the indirection to be performed automatically. Also, the overhead caused by the extra level of indirection is unacceptable in some cases. In the case of ports, for example, it significantly increases the cost of reading or writing a character, since these operations otherwise involve only two or three memory references. Finally, if a list of weak pointers is maintained (say to the set of objects in a large hash table or to a set of externally allocated objects), the entire list must be traversed to find the pointers that have been broken, even if none or only a few of the elements have been dropped by the collector. This is especially undesirable in a system with a generation-based collector, since some or all of the elements may be located in older generations not recently subject to collection.

A number of Lisp and Scheme systems contain a primitive finalization mechanism that is not made available to the user but is used internally by the runtime system. MultiScheme, T, and Chez Scheme (among others) use such a mechanism to finalize files. Chez Scheme also supports the elimination of unnecessary oblist entries, as proposed by Friedman and Wise [6].

Dickey [3] has proposed a user-level mechanism that allows a program to register objects for finalization. The procedure *register-for-finalization* accepts two arguments: an object and a thunk (zero-arity procedure). The thunk is invoked automatically during garbage collection if the object has been reclaimed. If implemented properly, this mechanism can eliminate the overhead of searching through a list of weak pointers. Since the object itself is not preserved, however, this solution suffers from the other problems associated with weak pointers. In addition, the thunk is not permitted to cause heap allocation since it is invoked as part of the garbage collection process and must not cause another garbage collection. This is an unfortunate restriction, both because it eliminates a useful set of tools and because it forces

[3] Weak sets were originally called populations.

the programmer to be aware of all sources of allocation, some of which may not be obvious. Furthermore, since garbage collections can happen at arbitrary times, the programmer has no control over when the actions are invoked. Errors that occur within the thunk are problematic as well; since they must not be allowed to prevent the invocation of other finalization thunks, error signals must be suppressed or somehow delayed until all finalization is complete.

A discussion of various finalization mechanisms found in other languages and operating systems such as object destructors in C++, final actions for modules in Euclid, and finalization actions for limited types in Ada 9X, can be found in [7]. None of the mechanisms described there, however, provide a general solution to the problems mentioned in Section 1.

3 Guardians

Guardians are created using the zero-arity primitive *make-guardian*:

$$(\textit{make-guardian}) \rightarrow \langle \textit{guardian} \rangle$$

A guardian is represented by a procedure that encapsulates a group of objects *registered* for preservation. When a guardian is created, the group of registered objects is empty. An object is registered with a guardian by passing the object as an argument to the guardian:

```
> (define G (make-guardian))
> (define x (cons 'a 'b))
> (G x)
```

The group of registered objects associated with a guardian is logically subdivided into two disjoint subgroups: a subgroup that we shall refer to as "accessible" objects, and one that we shall refer to as "inaccessible" objects. Inaccessible objects are objects that have been *proven* to be inaccessible (except through the guardian mechanism itself), and accessible objects are objects that have not been proven so. The word "proven" is important here; it may be that some objects in the accessible group are indeed inaccessible, but that this has not yet been proven. Depending upon the implementation, this proof may not be made in some cases until long after the object actually becomes inaccessible.

Objects registered with a guardian are initially placed in the accessible group, and are moved into the inaccessible group at some point after they become inaccessible. Objects in the inaccessible group are retrieved by invoking the guardian without arguments. If there are no objects in the inaccessible group, false (#f) is returned. Continuing the above example:

```
> (G)
#f
> (set! x #f)
...
> (G)
(a . b)
> (G)
#f
```

The initial call to G returns #f since the pair bound to x is the only object registered with G, and the pair is still accessible through that binding. At some point after this binding is nullified, however, the object shifts into the inaccessible group and is therefore returned by the later call to G.

Although an object returned from a guardian has been proven otherwise inaccessible, it has not yet been reclaimed by the storage management system and will not be reclaimed until after the last reference to it within or outside of the guardian system has been dropped. In fact, objects that have been retrieved from a guardian have no special status in this or in any other regard. This feature circumvents the problems associated with finalization of shared or cyclic objects. A shared or cyclic structure consisting of inaccessible objects is preserved in its entirety and each piece registered for preservation with any guardian is placed in the inaccessible set for that guardian. The programmer then has complete control over the order in which pieces of the structure are processed.

An object may be registered with a guardian more than once, in which case it is retrievable more than once:

```
> (define G (make-guardian))
> (define x (cons 'a 'b))
> (G x)
> (G x)
> (set! x #f)
...
> (G)
(a . b)
> (G)
(a . b)
```

It may also be registered with more than one guardian:

```
> (define G (make-guardian))
> (define H (make-guardian))
> (define x (cons 'a 'b))
> (G x)
> (H x)
> (set! x #f)
...
> (G)
(a . b)
> (H)
(a . b)
```

One can even register one guardian with another:

```
> (define G (make-guardian))
> (define H (make-guardian))
> (define x (cons 'a 'b))
> (G H)
> (H x)
> (set! x #f)
> (set! H #f)
...
> ((G))
(a . b)
```

(Of course, the last expression is dangerous, since there is no guarantee that (G) will not return #f.)

At what point does an inaccessible object become available for retrieval from a guardian? In general, the storage management system responsible for reclaiming the storage from inaccessible objects is also responsible for moving otherwise inaccessible objects from the accessible group to the inaccessible group. In a system such as ours that employs a garbage collector to reclaim inaccessible objects, the garbage collector maintains a list of registered objects with their associated guardians. This list is traversed after collection and any objects that have not been marked or forwarded are forwarded at that time (saved from destruction) and placed into the inaccessible group.

The example below demonstrates how guardians may be used in Scheme to ensure that dropped ports are closed. New "guarded" open operations are defined in terms of the existing operations (open-input-file and open-output-file), and a new exit procedure is defined in terms of the existing exit procedure.

```
(define port-guardian (make-guardian))

(define close-dropped-ports
  (lambda ()
    (let ([p (port-guardian)])
      (if p
          (begin (if (output-port? p)
                     (begin (flush-output-port p)
                            (close-output-port p))
                     (close-input-port p))
                 (close-dropped-ports))))))

(define guarded-open-input-file
  (lambda (pathname)
    (close-dropped-ports)
    (let ([p (open-input-file pathname)])
      (port-guardian p)
      p)))

(define guarded-open-output-file
  (lambda (pathname)
    (close-dropped-ports)
    (let ([p (open-output-file pathname)])
      (port-guardian p)
      p)))

(define guarded-exit
  (lambda ()
    (close-dropped-ports)
    (exit)))
```

In this implementation, dropped ports are closed whenever an open operation is performed or upon exit from the system. In many Scheme and Lisp systems it is possible to cause the garbage collection handler to perform arbitrary actions after collection completes (with the caveats mentioned in Section 1); in such systems it may make sense to cause *close-dropped-ports* to be invoked after collection instead. In Chez Scheme, a program does this by installing a new "collect-request" handler:

```
(collect-request-handler
  (lambda ()
    (collect)
    (close-dropped-ports)))
```

Guardians are also useful in conjunction with weak pairs. Weak pairs are like normal pairs except that the car field of a weak pair is a weak pointer, as described in Section 2. Weak pairs are created using *weak-cons* and manipulated using normal list processing operations, car, cdr, pair?, map, *etc.*[4] The existence of a weak pointer to an object in the car field of a weak pair does not prevent the object from being transferred from the accessible list of a guardian to the inaccessible list, and the weak pointer is not broken when such a transfer is made.

Figure 1 contains a simple hash table implementation that demonstrates how guardians and weak pairs can be used together to allow removal of useless entries. Support for removing useless entries is entirely contained within the shaded areas of the figure. When a *key/value* pair is added to the table, *key* is registered with a guardian associated with the hash table. This guardian is checked for keys to remove each time the hash-table access procedure is called. Since the *key/value* pair is a weak pair, the pointer to *key* is weak and does not prevent *key* from being transferred to the inaccessible list of the guardian.

Many Scheme and Lisp systems provide "eq" hash tables. Eq hash tables permit arbitrary objects to be used as keys with fast hashing based on the virtual memory address (the name "eq" comes from the name of the address equality predicate eq?). Since an object may be

[4] Some Scheme and Lisp systems have a distinct weak-pair type and related operations such as *weak-car* and *weak-cdr*.

```
(define make-guarded-hash-table
  (lambda (hash size)
    (let ([g (make-guardian)] [v (make-vector size '())])
      (lambda (key value)
        (let loop ([x (g)])
          (if x
              (let ([h (hash key size)])
                (let ([bucket (vector-ref v h)])
                  (vector-set! v h (remq (assq x bucket) bucket))
                  (loop (g))))))
        (let ([h (hash key size)])
          (let ([bucket (vector-ref v h)])
            (let ([a (assq key bucket)])
              (if a
                  (cdr a)
                  (let ([a (weak-cons key value)])
                    (g key)
                    (vector-set! v h (cons a bucket))
                    value))))))))))
```

Figure 1. *make-guarded-hash-table* accepts a hash procedure and a table size and returns a hash-table access procedure. The access procedure accepts a key and a value. If the key is already present in the table, the existing value is returned; otherwise, the key is added to the table along with the value provided. Sometime after a key becomes inaccessible it is returned by the guardian *g*, and the corresponding key-value pair is removed from the table. The definition of an "unguarded" hash table is obtained by deleting the shaded areas.

moved during a garbage collection, however, its address and hence its hash value may change. This problem is often solved by rehashing such tables after a collection or, more commonly, after a lookup has failed following a collection. In a generation-based collector much of this work is wasted for keys that are no longer forwarded during every collection because they have survived long enough to have advanced to older generations.

One solution to this problem is to use a "transport guardian" that returns an object when it has been moved (transported) rather than when it has become inaccessible. The system could then rehash only those objects that have been moved since the last rehash.

A useful conservative form of transport guardian may be implemented in terms of ordinary guardians and weak pairs. A conservative transport guardian returns all objects that have moved but may also return some objects that have not moved.

The code for implementing conservative transport guardians is given below. The approach is to allocate a fresh "marker" that is guaranteed to be no older than the object to be guarded (since the marker is newly allocated), register the marker with a guardian, and drop the reference to the marker so that it will be returned by the guardian after any collection to which the marker has been subjected. When the marker is returned by the guardian, the object may also have been subject to the same collection and thus is returned by the transport guardian. Since the same marker is re-registered with the guardian each time it is returned, it will gradually "age" along with the object providing the desired "generation-friendly" behavior. In order to prevent the transport guardian from holding onto an otherwise inaccessible object, the marker is a *weak-pair* whose car field contains the object.

```
(define make-transport-guardian
  (lambda ()
    (let ([g (make-guardian)])
      (case-lambda
        [(x) (g (weak-cons x '*))]
        [() (let loop ([m (g)])
              (and m
                   (if (car m)
                       (begin (g m) (car m))
                       (loop (g)))))]))))
```

4 Implementation

Adding guardians and weak pointers to our generation-based garbage collector was surprisingly straightforward. This section describes briefly the basic collection

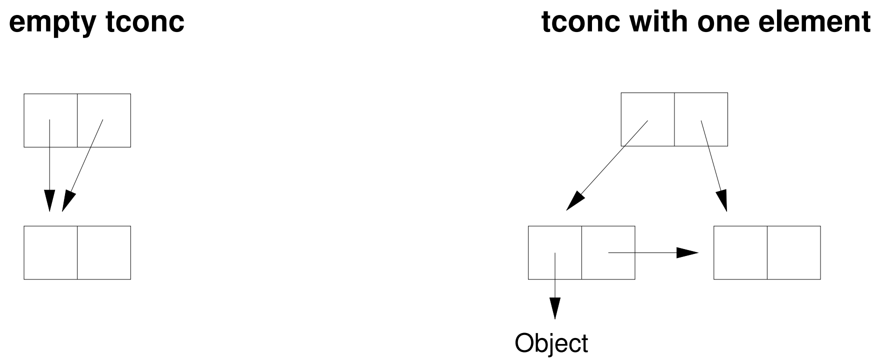

Figure 2. An empty tconc and a tconc with one element are shown above. Empty cells represent "don't care" values; neither collector nor mutator references such cells.

algorithm and the modifications necessary to incorporate guardians and weak pointers.

The number of generations and the promotion and tenure strategies supported by the collector are under programmer control. In order to simplify the discussion, however, we assume a fixed number of generations 0 through n (0 being youngest) with the following simple promotion and tenure strategy:

- New objects are placed in generation 0.
- Objects in generations less than or equal to g that survive a collection of generation g, $g < n$ are placed in generation $g+1$.
- Objects that survive a collection of generation n are placed in generation n.
- Generation 0 is collected each time there is a collection; older generations are collected less frequently (the older the generation, the less frequently it is collected).
- When a generation is collected, all younger generations are collected as well.

The generation into which objects are copied during a particular collection is referred to as the target generation. The collector performs a stop-and-copy collection from the generations being collected into the target generation.

The actual guardian interface described in Section 3 is a packaging of a lower-level interface. In the low-level interface, the garbage collector maintains a *protected list* of object/guardian pairs for each generation. Each time an object is registered with a guardian, a new pair (of the object and guardian) is added to the protected list for generation 0. After a collection of generation g, each element in the protected list of each generation i, $i \leq g$ is visited (the protected lists themselves are *not* forwarded during collection). If the object has been forwarded, it must be accessible (via a nonweak pointer) outside of the protected list, and the object/guardian pair is placed in the protected list of the target generation. If not, it is placed in the inaccessible group of the guardian and dropped from the protected list. In either case, both the element and the guardian are forwarded.

Although guardians are procedures at the user level, internally they are represented as a form of queue called a *tconc* (the name comes from an old Lisp construct of the same name). A tconc consists of a list and a header; the header is an ordinary pair whose car field points to the first cell in the list and whose cdr field points to the last cell in the list (see Figure 2). In our mechanism, the tconc representing a guardian holds the list of inaccessible objects; the garbage collector adds elements to the end of this list while the mutator removes elements from the front of the list.

We have chosen to use the tconc representation and designed the protocols for manipulating the tconc so that critical sections are unnecessary in both the mutator and collector (see Figures 3 and 4). Since the collector cannot be interrupted by the mutator in our current implementation, we do not presently rely on the fact that the collector does not require critical sections; however, since the collector can interrupt the mutator at any point, we do avoid the need for a critical section in the guardian code that returns inaccessible objects to the mutator.

An empty tconc is one in which both fields of the header point to the same pair; what the fields of this pair contain is unimportant. The mutator is permitted to manipulate the car field of the header; it is also allowed to compare the car field with the cdr field to determine if the tconc contains any elements. The collector is permitted to manipulate the cdr field of the header and the pair to which the cdr field points. In order to avoid the need for critical sections, the collec-

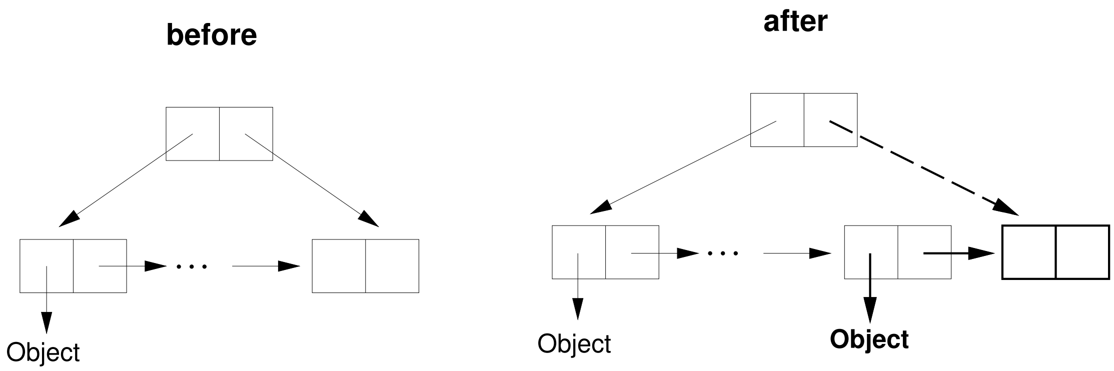

Figure 3. The collector adds new elements to the tconc by modifying the car field of the old last pair in the list to point to the new element and the cdr fields of both the old last pair and the header to point to a new last pair. Emboldened objects and pointers represent updated or new pieces, and the dashed pointer represents the final update.

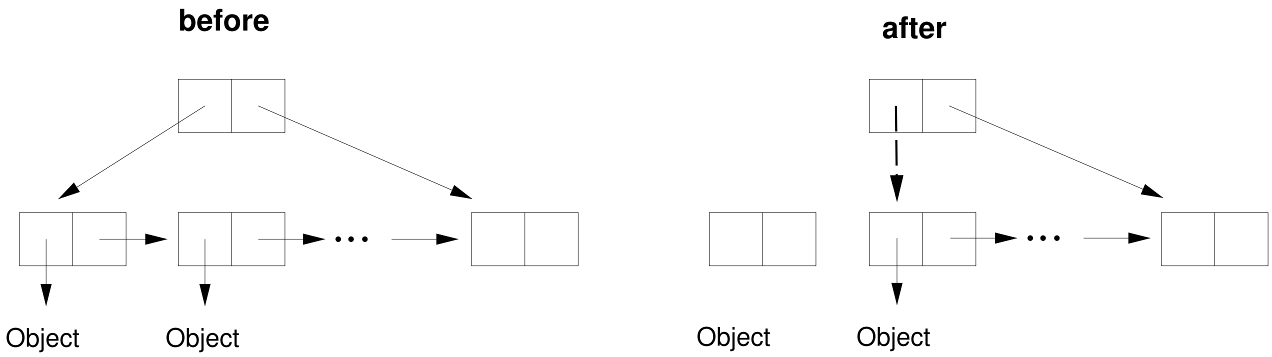

Figure 4. The mutator retrieves elements from the tconc by modifying the car field of the header to point to the second pair in the list.

tor adjusts the cdr field of the header last so that the mutator does not see that there is a new element until the element is fully installed.

In Figure 4, the pair that had been pointing to the first element is shown (after the operation) with "don't care" values in its car and cdr fields. Semantically, it is not necessary to remove the pointers that had been contained there. However, since the pair is sometimes in an older generation than the objects to which it points, maintaining these pointers after they are no longer needed may result in unnecessary storage retention.

The Scheme code that packages up the tconc structure is shown below. The syntactic form **case-lambda**, a multi-interface, multi-body version of **lambda** [5], is used to construct the procedure representing the guardian. The tconc structure is created outside of the **case-lambda** and is visible inside via the name *tc*. The procedure *install-guardian*, which is an internal routine provided by the collector module, simply adds its argument to the protected list for generation 0.

```
(define make-guardian
  (lambda ()
    (let ([tc (let ([x (cons #f '())]) (cons x x))])
      (case-lambda
        [() (and (not (eq? (car tc) (cdr tc)))
                 (let ([x (car tc)])
                   (let ([y (car x)])
                     (set-car! tc (cdr x))
                     (set-car! x #f)
                     (set-cdr! x #f)
                     y)))]
        [(obj) (install-guardian (cons obj tc))]))))
```

The portion of the collection algorithm that handles protected lists is described in pseudo-code below. It follows the algorithm as sketched above, but is complicated somewhat by the need to discard pairs from the protected lists when the corresponding guardian (tconc) is itself no longer accessible. Otherwise, all objects registered at the time the guardian is dropped can be reclaimed only after they have all become inaccessible.

```
pend-hold-list := pend-final-list := empty
For each generation i from 0 to g
  For each (obj . tconc) pair in protected[i]
    If forwarded?(obj)
      move (obj . tconc) to pend-hold-list
    Else
      move (obj . tconc) to pend-final-list
  End For
  protected[i] := empty
End For

Loop
  final-list := empty
  For each (obj . tconc) pair in pend-final-list
    If forwarded?(tconc)
      move (obj . tconc) to final-list
  End For
  If empty?(final-list) Exit Loop
  For each (obj . tconc) pair in final-list
    forward(obj)
    tconc := get-fwd-addr(tconc)
    add obj to the tconc
  End For
  kleene-sweep(g)
End Loop

For each (obj . tconc) pair in pend-hold-list
  If forwarded?(tconc)
    tconc := get-fwd-addr(tconc)
    obj := get-fwd-addr(obj)
    move (obj . tconc) to protected[target-generation]
  End If
End For
```

In the code above the predicate forwarded?(*obj*) is true when *obj* has been forwarded during this collection or when it resides in a generation older than those being collected. Similarly, get-fwd-addr(*obj*) returns either the forwarding address of *obj* or the address of *obj* itself. The procedure kleene-sweep(g) iteratively sweeps copied objects until there are no newly copied objects to sweep.

The first block of code separates accessible objects from inaccessible objects, placing the former (along with their *tconcs*) onto the *pend-hold-list* and the latter onto the *pend-final-list*. The second block of code iterates over the *obj*/*tconc* pairs in *pend-final-list*; if the *tconc* is accessible, *obj* is forwarded and added to the *tconc*. If the *tconc* is not accessible, it may become accessible during the sweeping phase (if pointed to from within one of the *obj*s) and is therefore left on *pend-final-list* for the next iteration. The third block moves each *obj*/*tconc* pair whose *tconc* has survived from *pend-hold-list* to the protected list of the target generation.

Weak pairs are supported as follows. Chez Scheme employs a segmented memory system in which the heap is structured as a set of segments (each currently 4K bytes in size). Each segment belongs to a specific space and generation; the space and generation to which each segment belongs is maintained in a segment information table with one entry per segment. The segments that comprise a space or generation are generally not contiguous. This arrangement has many benefits, including the ability to segregate objects based on their characteristics, such as whether they are mutable or whether they contain pointers [4]. We use this ability in the implementation of weak pairs, which are always placed in a distinct "weak-pair" space. When pairs found in the weak-pair space are traced during the normal garbage collection, they are treated like normal pairs except that the car field is not touched. A second pass through the weak-pair space is made after garbage collection; during this second pass, if the object pointed to by the car field of a weak pair has been forwarded, the new address is placed in the car field of the weak pair. Otherwise, #f is placed in the car field. The second pass through the weak-pair space occurs after the garbage collector has handled the protected lists (including the forwarding which is done there), so if the car field of a weak pair points to an object that has been salvaged, the object will still be in the car field after collection. Only when there are no pointers outside of the weak-pair space is the car field set to #f.

5 Conclusions

In this paper we have described *guardians*, which are entities that abstract the process of saving objects from destruction by the garbage collector in order that clean-up actions can be performed using the data stored within the objects. The program retains full control over when clean-up actions are performed, eliminating the need for some critical sections, restrictions on the use of allocation within the clean-up actions, and problems with the order in which pieces of shared or cyclic structures are processed. Unlike many previous mechanisms, the guardian mechanism actually preserves the object; most other mechanisms either discard the object and leave behind a flag or discard the object and automatically invoke some clean-up action associated with the object. Several problems with other mechanisms are avoided in this manner. The guardian mechanism, by itself or in combination with weak pairs, can be used to solve a variety of problems, from closing dropped ports to removing unused entries from hash tables. Our implementation is "generation-friendly" in the sense that the additional overhead within the generation-based garbage collector is proportional to the work already done there, and the overhead within the mutator is proportional to the number of clean-up actions actually performed.

We have considered a slightly more general guardian interface, in which the guardian accepts an "agent" in addition to the object when an object is registered for preservation. Rather than returning the object when it becomes inaccessible, the guardian returns the agent. Since the agent can be the object itself, this subsumes the simpler interface. The primary benefit of this change is that it allows objects to be discarded if something less than the object is needed to perform the finalization, although the agent might actually contain more than just what is contained within the object or something altogether different. We have not yet determined the full impact of this change on the collector.

For the applications we have examined, we have found that neither weak pairs nor guardians satisfactorily replaces the other. On the contrary, we have found that not only are both mechanisms useful but that they both work well together, as evidenced by the ease with which guarded hash tables and transport guardians are implemented.

Although we have designed and implemented the guardian mechanism for use in Scheme, there is nothing about the mechanism that is particular to the Scheme language. Adapting the mechanism to other languages and collection strategies should be straightforward.

Acknowledgements: We wish to thank David Wise for providing comments on an earlier draft of this paper and Mike Ashley for an insight that led to a simpler transport guardian implementation.

References

[1] Martin C. Atkins. *Implementation Techniques for Object-Oriented Systems.* PhD thesis, University of York, 1989.

[2] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[3] Ken Dickey. private communication.

[4] R. Kent Dybvig, David Eby, and Carl Bruggeman. Flexible and efficient storage management using a segmented heap. *in preparation.*

[5] R. Kent Dybvig and Robert Hieb. A new approach to procedures with variable arity. *Lisp and Symbolic Computation*, 3(3):229–244, September 1990.

[6] Daniel P. Friedman and David S. Wise. Garbage collecting a heap which includes a scatter table. *Information Processing Letters*, 5(6), December 1976.

[7] Barry Hayes. Finalization in the collector interface. In *Proceedings of the International Workshop on Memory Management IWMM92*, pages 277–298, St. Malo, France, September 1992. Springer-Verlag.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., second edition, 1973.

[9] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[10] James S. Miller. *MultiScheme: A Parallel Processing System Based on MIT Scheme.* PhD thesis, Massachusetts Institute of Technology, September 1987.

[11] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual*, fourth edition, September 1988.