Finalization in the Collector Interface

Barry Hayes

bhayes@cs.stanford.edu Stanford University, Department of Computer Science, Stanford, CA 94309, USA

Abstract. When a tracing garbage collector operates, it treats the objects and pointers in the system as nodes and edges in a directed graph. Most collectors simply use the graph to seek out objects that have become unreachable from the root objects and recycle the storage associated with them.
A few collector designers have hit on the idea of using the trace to gather other information about the connectivity of the graph, and notify user-level code when an object is reachable from the roots, but only in a restricted way. The user-level code typically uses this information to perform some final action on the object, and then destroys even the restricted access to the object, allowing the next pass of the garbage collector to recycle the storage.
Finalization is useful for appropriating the power of garbage collection to manage non-memory resources. The resource in question can be embodied in a memory object with finalization enabled. When the memory resource is reachable only through restricted paths, the non-memory resource can be recycled and the restricted access destroyed. The users of the resource need not coordinate to manage, nor do they need to know that the resource is precious or needs finalization.
This paper presents system-level details of five different implementations of finalization in five different systems, and language-level details of several languages that have defined similar mechanisms. These comparisons highlight several areas of concern when designing a system with finalization.

1 Introduction

Garbage collection is sometimes trying to serve two antithetical goals. First, some languages and systems see collection solely as a way to make a finite memory resource appear larger. The programmer need not worry about memory because there is a large supply, and the garbage collector helps maintain the fiction. In this role, the collector must be invisible, lurking in the shadows, and no side effects of collection should be apparent to other code in the language.

Other languages and systems see garbage collection as a valuable opportunity to learn more about the connectivity of objects, and try to make the information gleaned by the garbage collector available to other code. The most common use of this information is to implement weak or soft pointers. Another is to drive *finalization.* Finalization takes many varied forms, but the goal is to communicate information about the connectivity of objects from the garbage collector to other elements of the system.

With finalization this connectivity information is used to let a module that manages a resource know when no module other than itself has any remaining pointers to an object in its resource pool. When it knows this, it can invoke code on the object to do any clean-up that might be required, and return the resource to the pool. Without this connectivity information, the users of such a resource are required to cooperate in the management of the object, and the code required can be difficult to write, verify, and maintain. Making the connectivity information available allows the resource to be managed in a simple way, and lends the power of garbage collection to the management of other resources.

2 A Short History

Finalization seems to have grown from two different roots in computer science: the desire to have soft pointers for ease in engineering, and the desire to do correctness proofs in the presence of exception handling.

Soft pointers, also called weak pointers, are pointers that are not traced or counted by the garbage collector. Typically, when the collector notes that there are no hard pointers to an object, it collects the storage associated with the object and sets the soft pointers to a known value, often zero or NIL[1]. Soft pointers allow a process to monitor an object and know if it has been collected without interfering with the collection of the object.

Closely related to soft pointers are *populations*. A population is a clever kind of hash table — a "key" can be used to find a "value". Often these keys and values are simply objects, and the address of the key is hashed to find the location of the key/value pair. But when all other references to the key have vanished from the system, it will never be used to look up the associated value. A population differs from a simple hash table in that it is in bed with the collection system and does not allow this reference to retain the key[2]. Populations exist in many modern Lisp system.

The other related concept, error recovery, is particularly relevant to systems where the central focus is more on the data types than on the code, and where correctness concerns are important. Many languages include a facility whereby a block of code can have an attached clause that is executed if the block terminates

[1] One of the earliest soft pointer implementations was Interlisp-D's XPOINTERs [Xer85]. It did not change the values of soft pointers, but would just deallocate the object. Users of soft pointers could have all the problems associated with dangling references that garbage collection was supposed to have solved for them. Soft pointers were one of the aptly-named "unsafe" features of the language.

[2] If the garbage collector is also copying objects, the address of the object will be changing from time to time, and that too provides motivation to make the garbage collector and the population implementation interconnected.

abnormally. This is sometimes called "unwind protection," since it protects the block in question from the call stack unwinding that occurs automatically when an error throws control from the location of the error to a handler for that error. The unwind protection code is expected to take any necessary activity to clean up after the error-exit, and maintain any invariants needed in the program. Any program using semaphores, for example, benefits from unwind protection, in that an error between the points where the resource is locked and unlocked could otherwise cause the resource to remain locked. The unwind protection code can clean up after the error, and might be expected to return the resource to a consistent state and unlock it.

Often, the invariants are more closely associated with the data types than with the code, and a correct program would have nearly the same unwind protection associated with every block that declared an instance of that type. For example, if a block declares a file, it might be expected that when the block is exited, either normally or because of an error, the file's buffers will be flushed, and the file will be closed. It would be perfectly acceptable to include the code to do this in an unwinding clause of every block that declared a file, but for two things: programmers would invariably miss a few, leading to subtle bugs, and the code would be less readable for the constant clutter. Instead, the declaration of the type can be extended with what is in essence the common unwind clause, and each block containing a declaration can be assumed to have such a clause.

This type-centered formulation of final action extends to dynamically allocated objects as well. When an object is about to be freed, either explicitly or by a garbage collector, the same unwind phrase can be run. All instances of the type, allocated on the stack or heap, receive the same final treatment, and have a chance to correct any invariants before they are returned to storage. C++ destructors are the best known exemplar of this style, but C++ has no native garbage collection, and only experimental exception handling.

The issues involved for finalization of stack variables center around exceptions and error recovery [SMS81], and the issues involved for finalization of heap objects center around the topology of the connections between objects.

3 Survey of Finalization

This section is a survey of systems where finalization is available. Where I have been able to find out details about how the collection system works, I have presented as complete a description of the system as I can. Previous work [AN88] has identified a set of properties that might be desired from finalization. Some systems I know only through language reference manuals and reports, and for these systems the summary is often quite brief. I encourage anyone with knowledge of other finalization systems or more complete knowledge of any of these systems to contact me.

3.1 Lisps

Almost every Lisp dialect has some form of hash tables, and a few have populations that garbage collect inaccessible keys.

Scheme allows files to be closed automatically provided "it is possible to prove that the [file] will never again be used for a read or write operation." [AAB+91, Section 6.10.1] The garbage collector can be seen as constructing such a proof.

T [RAM84], a Lisp variant influenced by Scheme, has finalization for files but for no other data types. Files are a highly trusted client of the collector, and the collector explicitly calls a file routine to close all inaccessible files near the end of the collection. The files are known to be inaccessible by use of T's extensive weak pointer system.

I have heard rumors that other Lisp implementations have similar finalization hooks for trusted clients, but have been unable to track down any definitive sources.

3.2 Sun NeWS

The NeWS package from Sun is a windowing system using a liberally extended PostScript, and includes a conceptually parsimonious finalization interface [Sun90]. There are two operations on pointer values, *soften* and *harden*. By default, a pointer value is hard, but these operations take a pointer value of either firmness and turn it into the firmness desired. A third operator, *soft*, queries the firmness of a pointer without changing it.

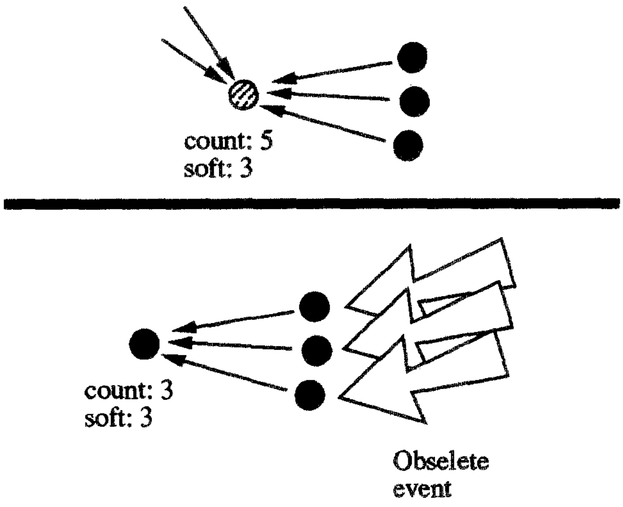

Fig. 1. NeWS Finalization

The garbage collector counts references, and maintains both a total count and a count of the soft references for each object[3]. Both reference counts are updated as needed every time a pointer is changed — they are always accurate between execution of any two PostScript operators.

Whenever the counts of total references and soft references become equal and are not both zero, the system generates an *Obsolete* event for that object. This can happen only if a hard reference is deleted or made soft. It is expected that every holder of a soft pointer will have expressed interest in the event.

3.3 Euclid

Euclid allows a module, implementing an abstract data type, to "include an *initial* action which is executed whenever a new variable of the module type is created, and a *final* action which is executed whenever such a variable is destroyed." [LHL+77, page 22] If several module variables are declared, they are initialized in order of declaration and finalized in reverse order. This is to allow later-declared objects to access fully initialized, previously declared objects at initialization, and to guarantee that at finalization no object will attempt to access a finalized object.

Euclid requires that initialization and finalization also run when an object is explicitly allocated and freed. Presumably, the finalization code would also run if the object is implicitly deallocated by garbage collection, but the definition does not make this clear.

A sticky point comes up when trying to finalize dynamically allocated objects in a sensible order. The system would like to guarantee that no object would have any methods invoked on it after it has been finalized, but two or more objects may cyclicly reference one another. If the collector finalizes one of the objects in the cycle, it may still be reachable from another that requires access to the now finalized object. There is no information available that would allow the collector to choose objects to finalize wisely under these conditions. Finalization order of cyclic structures is a problem in other languages, and will be examined in Section 4.4.

3.4 C++

The C++ language is not defined to have a garbage collector, but has *constructors* and *destructors* quite similar to the concepts found in Euclid [ES90]. The destructor is a method of an object type, and will be called by the system when the storage for an instance of that type is about to be returned to the system. It will be called as a consequence of explicitly *deleting* the object, if the object is on the heap, and it also will be called when a block declaring the object is exited, either normally when

[3] There is also a third class of references, *uncounted*, that is not available to the user, but is used by the system to break cycles among its structures.

the evaluation of the block is finished, or abnormally when the block is exited with a break, continue, return, or goto.

Within a block the order of construction and destruction is defined just as in Euclid: in declaration order for construction, and in reverse order for destruction. Types in C++ have multiple inheritance, and so the initializers and finalizers for each of the base classes, if any, have to be run at construction and destruction of objects. "[To initialize a class object] the base classes are initialized in declaration order [...], then the members are initialized in declaration order [...], then the body of [the initializer] is executed. The declaration order is used to ensure that sub-objects and members are destroyed in reverse order of initialization." [ES90, page 292]

One problem with C++ destructors stems from compiler-generated temporaries. Compiler-generated temporaries have no obvious scope, and so it is not clear when to run the destructor method. Adding multiple threads of control to C++ in the presence of destructors may also prove difficult, since pointers to objects may be passed out of the static scope where the object is created.

There have been several proposals to date for adding garbage collection to C++ [Bar89, Ede90, Det91]. One of these [Det91] explicitly disables destructors due to worries about compatibility. This is correct if the only purpose of the destructor is to explicitly delete other objects it references — the collector will do just that without any help — but will fail if the destructor has other effects.

3.5 Modula-3

Modula-3 has garbage collection without finalization, but extensions have been proposed [Hud91]. This proposal allows destructors similar to C++ and after each collection invokes the destructors for the unreachable objects in order from youngest to oldest. This is the same order they would be invoked in if the objects were stack-allocated, but the problems in using this order of finalization for heap-allocated objects is not addressed by the proposal.

Most of these problems occur when the objects form cycles of reference, and it seems reasonable that finalization should take the topological order, rather than the chronological order, of the objects into account when ordering finalization. This problem will be discussed in more detail in Section 4.4, and is common to almost all implementations.

3.6 Ada 9X

Ada has no finalization, but the Ada 9X revision does [DoD91b, Section 7.4.6]. Finalization is available for *limited types*, a restricted abstraction where assignment is not defined. Ada disallows objects of limited types in contexts where implicit assignment or copy would be needed, and so avoids any problems that arise in finalization of temporary values [DoD91a, Section 3.2.3.1]. Finalization actions occur when the scope of the program unit finishes, for static variables, and when objects are explicitly deallocated, for allocated variables.

In addition, packages [DoD91b, Section 7.4.6] have a form of finalization. When a generic package has an *exit handler*, exiting the scope where the package is instantiated will cause the handler to run, and the package can take final actions.

The two methods are similar, but if coordination among objects of the same type is required at finalization, the use of limited types seems superior to exit handlers.

3.7 ParcPlace Smalltalk

Finalization in the Smalltalk system available from ParcPlace [Par90] is similar to the NeWS finalization, but is a more direct descendant of populations. There is a special type of array called a *weak array*, containing weak pointers; weak pointers are not available anywhere in the system except these weak arrays.

The garbage collection subsystem contains both a generational collector for young objects and an incremental collector for old objects. Both are tracing collectors — the generational collector is a copying scavenger and the incremental collector is trace and sweep.

Pointers from weak arrays are traversed last in garbage collection, and when an object is found to be inaccessible except through a weak array, the collection system frees the storage associated with the object and stores the value zero in any weak pointer that is a reference to a reclaimed object. The object is truly collected, and the zeroing guarantees that there will be no dangling references.

To give finalization information back to the user, this simple weak pointer scheme has been combined with a notification step much like the Obsolete event in NeWS. After each garbage collection, any weak array that has had a pointer zeroed is sent a "changed" message. The zeros in the array give the indices of elements that have been collected; it is the responsibility of the user's code to make sure that any data needed after the element is collected are present elsewhere, and that the "changed" message is propagated to the collected object's *executor*, which is expected to take the action needed to maintain the invariants.

Figure 2 shows an object held by a weak array, as well as other pointers. When the other pointers are deleted, the object does not go away immediately. When the garbage collector discovers that the object is reachable only via the weak array, the object is collected and the array notified.

The usual programming idiom is to make the executor a shallow copy of the object. This is a distinct object with identical values for all of its variables — it points to the same objects that the finalized object points to, and any information needed to preserve invariants should be designed into the shared parts of the structure, not the finalized object. By the time the executor gets control, the memory allocated to the finalized object has been freed, and the object is unavailable to the executor.

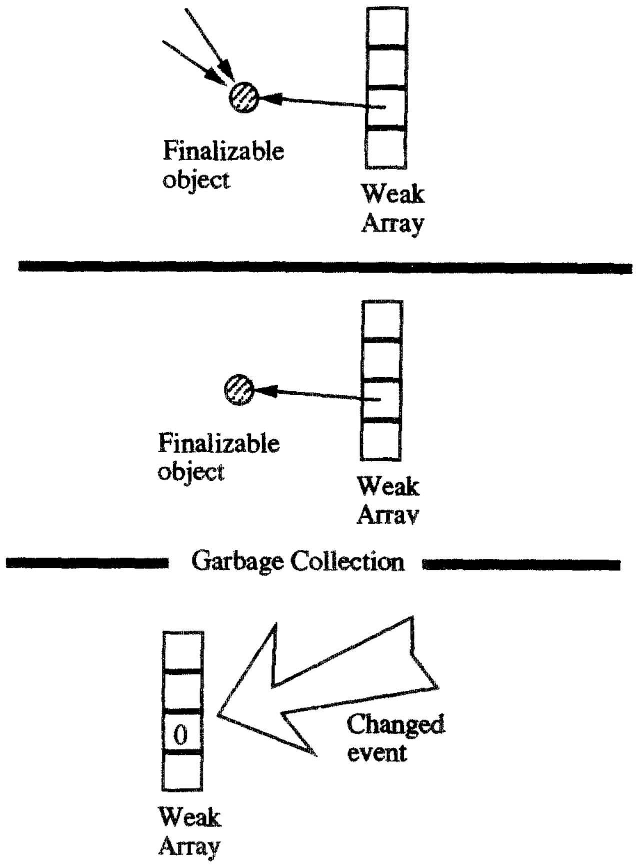

Fig. 2. Objectworks Finalization

3.8 AML/X

The object-oriented design language AML/X [NLT$^+$86] included a reference-counting garbage collection system and finalization package. One of the driving goals in the system was to study the interfacing between object-oriented systems and procedural systems. Procedural protocols involving return of resources were enforced by using finalization methods of objects [AN88, Atk89], but natural cycles in the objects prevented some of the finalizations from occurring.

In order to get more reliable finalization of cycles, the design was carried from the reference-counting collector to a tracing collector [AN88]. Each object has three bits, "mark," "destroy," and "colour." The mark bit is the usual tracing collection mark. The destroy bit shows if an object's finalization method must be or has been called. The colour bit is used to ensure that only one object in a cycle is finalized.

All objects ha' e the colour and mark bit initially set to zero, and collection is a five-phase process:

Mark Mark all objects reachable from the system roots, and reset the "destroy" bit on all reachable objects. Unreachable objects that were finalizaed by a previous collection may have the destroy bit still set.

Identify Candidates Examine each finalizable object in turn, setting the "destroy" bit in all unmarked, uncoloured, finalizable objects that do not have their destroy bit set, and setting the "colour" bit in them and all unmarked objects reachable from them. This phase colours objects that cannot be reached from the roots, but that are reachable from finalizable objects.

Prune If there are unreachable cycles of finalizable objects, at most one in each cycle should be selected. To do this, every finalizable object is visited in turn, and if it is coloured and its destroy bit is set, the destroy bit is cleared in all unmarked objects coloured in the previous phase. The destroy bit in the first object encountered in each cycle will not be reset unless the entire cycle is reachable from another finalizable object.

Scan Each allocated object that is neither marked not coloured is deallocated, and the mark and colour bits are cleared.

Finalize For every object with the "destroy" bit set, call its finalization method.

The designers of this system noticed a few flaws in it. First, the system's arbitrary choice of one object in a cycle can easily be wrong. It is difficult for users to predict the effects of finalization when cycles are involved. This is a problem inherent in cyclic finalizations, not a problem with this specific system.

Second, if an object's finalization method causes the object to become reachable from the roots, the "destroy" bit is still set and the finalization method will be called again on the next garbage collection. While an object from a resource pool may need to have its finalization called many times through its lifetime, it seems as if some kind of explicit "enable" call for finalization is needed to tell the difference between the return of an object to a pool, and the return of an object to a client.

The complexity of finalization in the tracing system, including the retracing of objects[4], drove one of the authors to consider other finalization techniques [Atk89]. The new proposal relies on weak pointers, much as the Objectworks system does.

[4] The P-Cedar system, outlined in Section 3.10 uses a different marking strategy and only a single mark bit to get almost exactly the same effect with none of the retracing.

Each finalizable object is paired with a *forwarding object.* All clients needing access to the finalizable resource are given pointers to the forwarding object instead, and all method calls to the forwarding object are passed to the client object — the forwarding object is invisible to the clients. The system maintains a weak pointer to each forwarding object, and so when the last client pointer is deleted the forwarding object is collected. The garbage collector then notifies a list of clients that a collection has occurred, and any finalizable object that has had its forwarding object collected can be finalized.

This system and the Objectworks system differ in the level that forwarding objects are defined — this system make them primitive, and Objectworks requires the users to roll their own or create variants. In addition, the propagation of the information that indicates that an object has been freed is more clearly defined in Objectworks. There does not seem to be an implementation of Atkins's system, and that allows many issues to remain unaddressed.

3.9 Cedar — The Early Years

D-Cedar[5], as implemented on the Xerox D-machines, uses a concurrent reference counting collector and a secondary trace and sweep collector [Rov85]. The reference counts are not always accurate for two reasons: references from the stacks are not counted, and the stacks are scanned conservatively. When an object's reference count goes to zero, it is placed in a special zero count table but not deallocated, since there may still be pointers to it from the stacks. Occasionally the garbage collector conservatively scans the stacks looking for bit patterns that, if they are pointers, point to objects with reference counts of zero[6]. In the end, any object that has a reference count of zero and is not pointed to from a stack is collected or finalized.

There is no finalization available for objects declared statically in Cedar, but typical programming practice is to explicitly create any objects needed in the block and assign them to local reference variables. Some time after the block exits, the collector will discover that the objects are no longer reachable, and they will be reclaimed. The order of initialization is under user control, and the order of finalization is determined by the topology of the interconnections among the objects.

[5] The Cedar system, including the Cedar language, has been implemented twice: the first implementation ran on Xerox's family of machines, the Dorado, the Dandelion, the Dolphin and the Daybreak. The second implementation was designed for portability, and currently runs on a number of standard platforms including Unix and Posix. The storage management has changed almost completely between the two implementations. To keep the discussion on an even keel, the first implementation will be called D-Cedar, for Dorado-Cedar, and the second P-Cedar, for portable Cedar.

[6] The implementation is more complex, in that all processes are halted just long enough for the stacks to be copied, and the conservative search for pointers occurs in these copies. The collector runs concurrently with all other active processes.

The model for finalization in D-Cedar is that a package will manage objects of a certain type, and will be responsible for maintaining any invariants associated with those objects. New objects of that type will only be created by calls to the package, and when the package returns an object it may still have several private pointers to that object. The clients need not do anything specific to manage the object, but when the clients destroy the last pointer to the object, the package, which still holds pointers to the object, should be notified that the clients can no longer use the object.

Cedar is a typed language, and finalization in D-Cedar is strongly linked to types. Associated with any finalizable type are a *finalization queue* and a positive number indicating the count of *package references*. Any particular object of a finalizable type can be explicitly enabled for finalization by a call to the storage manager. This call sets a bit in the object's header, and decrements the object's reference count by the package reference count. From that point, the object is reference counted normally.

When an object has a zero reference count and there is no pointer to it from the stack, it is freed if the finalization bit in that object is not set. If the bit is set, the collector clears the bit, sets the reference count for the object to be the package count, determined from the object's type, and adds the object to the finalization queue for that type, allowing the package to do whatever is required with the object to maintain its invariants[7].

In practice, the use of a type-wide count of package references proves to be fragile. The package must ensure that it has the same number of pointers to every object enabled for finalization, and the writing of packages where finalization is used is a delicate affair. Catastrophic failures occurs when a dropped package pointer makes a reference count negative.

Notice also that the object is changed from finalizable to not finalizable when the finalization is run. The object may be explicitly set finalizable again, setting the bit and reducing the reference count, but it is not automatic. This helps prevent errors where the finalization code runs for each garbage collection without making progress on finalizing the object. Instead, the code will be run once, and if the object is neither made reachable nor re-enabled for finalization, it will be collected.

[7] When the package count is zero, this is what is sometimes called *resurrection semantics*, since the object has no pointer to it, and yet the collector creates one to enqueue the object. If you feel uncomfortable with the idea that the collector is creating a pointer to an object after the user has discarded all the pointers to it, recall that the call to the collector to enable finalization allows the system to squirrel away a pointer to the object, and that it is this pointer that is used to enqueue the object. In fact, that pointer exists — it is simply compressed into a single bit in the header of the object.

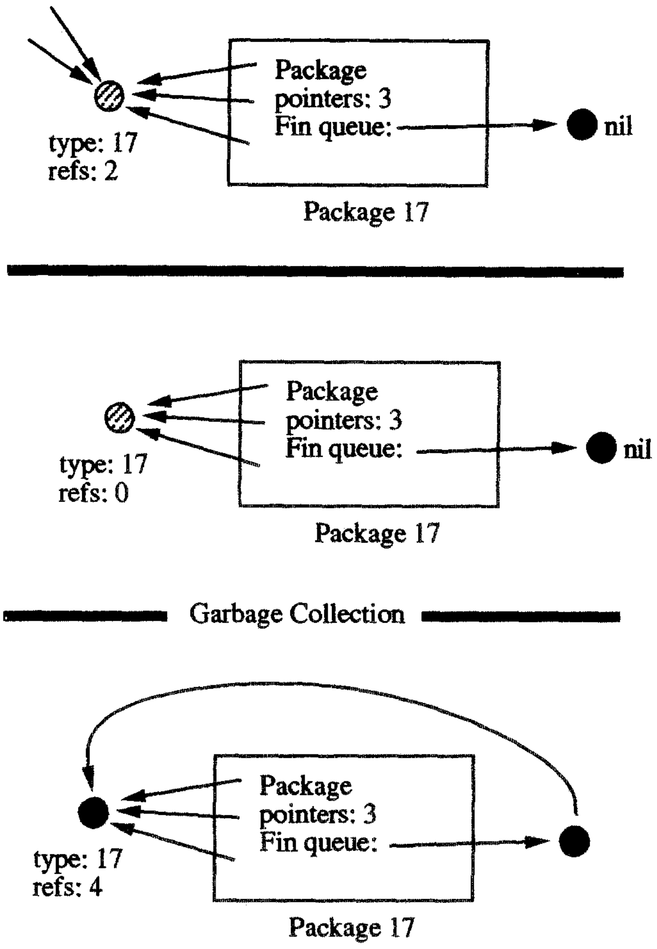

Fig. 3. D-Cedar Finalization

3.10 Cedar II — The Revenge

Some of the weaknesses in the storage management in D-Cedar were addressed in P-Cedar [ADH+89]. The reference counting collector was replaced with a conservative, generational, mark-and-sweep collector. This freed the programmer from the burden of breaking reference cycles in complex data structures[8].

[8] One of the common uses of finalization in D-Cedar was to break these cycles. For example, a tree where every leaf points back to the root will never be collected by simple reference counting — when the last reference to the root other than the leaves is deleted, the root will still have a non-zero count. A package count for the root would not make sense, since the count at the root is the number of leaves, and that is variable, rather than structural. But if a node is added above the root and all access takes place through that node, that extra node can be enabled for finalization. When all references to it are gone, the leaves

Most of the work of finalization has been put into a package distinct from the garbage collector. The finalization package is still tightly coupled to garbage collection, but the split seems valuable to insulate the two functions — collection and finalization — from each other.

To enable an object for finalization, a client calls a routine in the finalization package with a pointer to an object and a finalization queue. The package returns a pointer called a finalization handle. Strictly speaking, it is the handle itself, not the object, that is enabled for finalization. The only important operations available on a finalization handle are disable finalization, re-enable finalization, and dereference. The disable/re-enable calls do the obvious, and the dereference call returns a pointer to the object that was a parameter to the enable call that returned that handle. A disabled finalization handle functions simply as an indirect pointer to the object. The finalizer keeps the state needed to finalize objects; this is not the responsibility of its clients, and they often ignore the finalization handle returned, knowing that the finalization will occur nonetheless.

The collector traces from the roots, but does not trace through the finalization package's state to objects that are enabled for finalization[9]. At this point, any finalizable object that has been seen by the trace is accessible from the roots, and should not be finalized.

Only some of the objects unreachable from the roots are put on their finalization queues. The intent is to mimic the effects of the reference counting finalization of D-Cedar by only finalizing those objects that are not reachable from other finalizable objects. If P points to Q, and both are finalizable and unreachable from the roots, the system would like to finalize P. When P is put on its finalization queue, Q is now reachable from the roots via the queue, and should not be finalized.

The objects to be queued are found by another marking phase of the collector. It traces all of the pointers from unmarked finalizable objects, but does this without initially marking the finalizable objects themselves. After this marking is finished, any marked finalizable object was either marked by the first phase, and so is reachable from the roots, or was marked in the second phase, and so was reachable from a finalizable object[10]. Any unmarked finalizable object is reachable through neither

still point to the root, but there are no pointers to the uber-root. The finalization for the uber-root can walk the tree down to the roots and NIL the backpointers, allowing the reference counter to discover that the tree's storage can be reclaimed.

[9] The finalization package does not, in fact, keep pointers to the objects. The object pointed to by a handle contains a field that is a disguised copy of the pointer to the object, and the finalization package keeps a list of the currently enabled finalization handles. The collector does not recognize the disguised pointers as pointers when doing the trace.

[10] At the moment, a finalizable object that is reachable from itself but no other finalizable object is *not* finalized. This is considered to be a bug and will be changed. This might present a danger to an object that has access to an object of the same type as itself, since it might have to check the identity of the object against itself. For example, we

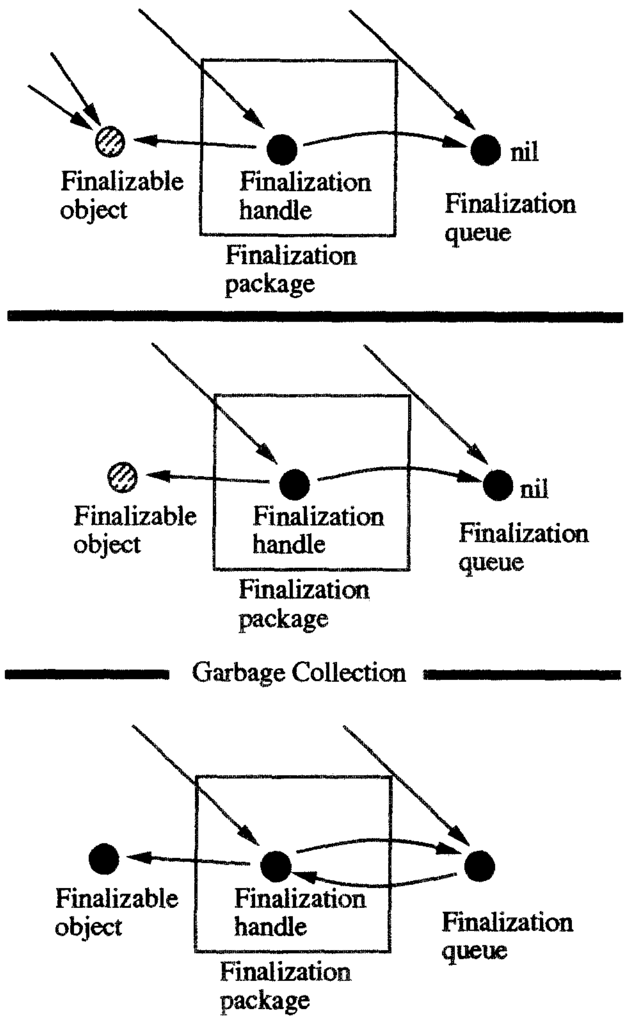

Fig. 4. P-Cedar Finalization

method. The finalization package can safely enqueue each unmarked finalizable object, and change its state to indicate that it has discharged its duty — the enqueued objects are no longer enabled.

might have a system that writes a record to a log file for each file finalized. Imagine the problems that might occur when the log file is closed.

4 Finalization Issues

As should be apparent, there is no consensus of opinion on how to design the interface between the collector and its clients to allow for finalization. There are at least four major decisions that a designer should consider in specifying a finalization interface: decoupling, promptness, locking, and cycles.

4.1 Decoupling

Aside from the Lisps that have a restricted group of clients for finalization, systems must be careful in how the results of tracing are communicated to the clients. If a collector were to directly call some kind of finalization routine for a client, it would risk aborting or looping the collection thread. Event and message systems have a natural way of dealing with this problem, since the messages provide a decoupling between the collector and its clients. Likewise, the Cedar queues let the collector enqueue an object without worrying about the consequences of calling general client code. Another option would be to fork a process from the collector for each client needing to be finalized.

4.2 Promptness

When a resource is recovered soon after it is no longer reachable from its clients, we say that it is *promptly reclaimed.* This measure is often applied to garbage collection, and it is also a useful measure of finalization. The range of promptness in the systems in Section 3 is broad.

The NeWS system recovers memory as soon as an object's reference count drops to zero, and sends Obsolete events as soon as the last hard pointer is deleted. Promptness is a benefit of being a reference counting collector. But objects that are unreachable from the roots and involved in cycles will never be recovered or finalized, and so these objects can hardly be said to be promptly reclaimed. Reference counting is both a boon and a bane for promptness.

Inexact collection, such as is found in generational collectors, has a similar effect. An object may be considered reachable because there is a pointer to it from an old object that is unreachable, but hasn't been collected yet. No object can guarantee that it will be promptly collected or finalized.

Conservative collection confounds the issue even more [BW88]. Not only can an object remain uncollected or unfinalized because of genuine pointers to it, but bit patterns that the collector treats as pointers are enough to keep objects uncollected. An application may earnestly exercise great care to NIL all of the pointers to an object to make finalization or collection occur, only to have an errant integer prohibit it.

Current collectors do not offer clients any options to help regulate the promptness of the service they get, and as object bases get larger and larger, generational collectors will make this problem worse and worse. It seems as if a good rule of thumb for finalization clients is to consider finalization to be a frill, and not to rely on it for promptness or correctness.

4.3 Locking

All too often, the garbage collector and finalization routines are overlooked as a source of parallelism in code, and parts of an aggregate structure will be finalized even when other clients might still be using it. For example, consider a tree with back-pointers from the leaves to the root, and an extra finalizable node at the top that will break all the cycles to aid a reference counting collector. Great care must be taken to ensure that this finalization does not occur while there is still a pointer to an internal node. The process holding that pointer might be surprised to find that the back-pointer is NIL.

Unfortunately, the simple ways of locking out the finalization are not guaranteed to work. For example, it might be thought that any process holding a pointer to the finalizable uber-root node while examining the other nodes would be safe, but compilers would look on the reference to the uber-root as dead, since it is not used, and might optimize away the load of the pointer. This is just another manifestation of collector/optimizer interactions, but lifted into the domain of finalization [Cha87, Boe91].

Neither of the obvious solutions to this problem are attractive. The optimizer can be prevented from performing some useful optimizations, but that's a performance cost many might blanch to pay. Modules could supply client calls to ensure that structures are not finalized when there is an active client, but this seems a violation of modularity, since it reveals to clients of the package that objects are finalized.

4.4 Cycles

When two or more finalizable resources are clients of one another in a cyclic order, finalization becomes much more complex. The system might decide to take a conservative view and finalize no object in the system. This guaranteed lack of promptness leads to resource leaks similar to memory leaks from cycles in a reference counting collector.

The system may try to guess an object in the cycle to finalize first, but if it chooses incorrectly, another object may try to make client calls to the first finalized object. The client of the finalized object may be unable to do anything reasonable now that the resource embodied in the first object has been rescinded [AP87].

Any system that chooses to finalize all the objects in a cycle in an unpredictable order dooms the programmer to adjudicating the correct order by use of mutual exclusion primitives. But this violates modularity, since each object must be aware of the cycles it might be involved in.

To break the symmetry of the cycle, some objects might use soft pointers to point to other objects to indicate that they do not require the softly-held resource at finalization, and would be willing to be finalized after the other resource. But this means that the softly-held resource might be finalized while its client is still firmly-held and active if the holder of the soft pointer is its sole client. It must be ready to have the resource finalized at any point, not only just prior to its own finalization.

Finalization of cyclic structures is a problem that the garbage collector cannot solve without further development on the interface between the languages and the finalization package. The current interfaces seem to be too narrow to address all of the situations that arise.

5 Conclusions

In many new systems and languages, garbage collection is considered indispensable. Programs can be crafted without worrying about memory leaks or dangling pointer bugs — the memory will be recycled when it is no longer needed and not before because the collector can guarantee when a memory object is no longer reachable from the roots.

If other system elements are allowed access to the information gathered by the garbage collector, they can make decisions about non-memory resource recycling, and allow these resources to be managed in much the same way as garbage collected memory.

Several recent researchers have tried to marry C++ destructors and garbage collection to get finalization, but previous efforts in finalization and the problems that have been encountered by users of finalization does not seem to be well known.

At least four systems, Objectworks, NeWS, D-Cedar, and P-Cedar, have been built with a garbage collector that allows user code some access to the information gathered by the collector. All four of these systems are in current use, but the finalization features seem to be little-known in the systems community, and even the memory management community.

6 Thanks

Many people helped with the gathering of information for this paper. Much of the leg-work for citations was done by Frank Jackson, who also checked that my description of ParcPlace Smalltalk was reasonably accurate. In that same vein, thanks to Stuart

Marks from Sun for the NeWS information, and Carl Hauser of Xerox PARC for the Cedar information.

This work was funded by Dr. John Koza, the Northern California Chapter of ARCS, Inc., and Xerox.

A A Garden of Delights

Many people have been active in the discussion of changes to finalization at Xerox PARC. The most active of these are Hans Boehm, Alan Demers, Carl Hauser, Christian Jacobi[11], and myself. There is a set of canonical examples we have been using in discussion of problems with and extensions to finalization. Much of the text of this section was provided by Carl Hauser; otherwise, it is hard to credit any particular example to any particular person.

A.1 File Descriptors

This example involves collection of non-memory, low level resources, and shows a simple case where finalization is valuable.

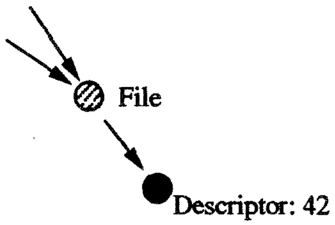

Fig. 5. Simple Finalization

Unix file descriptors are a resource that should be garbage-collected: to first approximation, if there are no copies of a file-descriptor in a program, then that file descriptor should be closed and freed. By allocating a memory object for each opened file descriptor, and uniformly passing the memory object around as the representative of the open file descriptor (instead of the file descriptor), we know to close the file descriptor when the memory object becomes unreachable.

A.2 Buffered File

In this example, two objects, each of which should logically be enabled for finalization, point to each other and form a cycle.

[11] Christian has shown the patience of a saint in letting us discuss the issues to death before we actually do anything that will let him fix the problem with his code.

Suppose we have an OpenFile abstraction, a BufferedFile abstraction and a Buffer abstraction. OpenFile objects contain file descriptors and hence need finalization enabled. They also refer to a Buffer object — the next buffer to be filled, for example. BufferedFile is one of perhaps many abstractions built on OpenFile, and each BufferedFile supplies the buffers used by its underlying OpenFile. BufferedFile's objects need finalization enabled to allow write buffers to be flushed prior to closing the underlying OpenFile object. The Buffer abstraction provides a field in each object for recording the owner of that buffer object so that low-level system interrupts that refer to memory locations in the buffer can be correctly forwarded to the proper OpenFile.

Since the Buffers are owned by BufferedFiles, this gives us a cycle containing two finalizable objects: a BufferedFile points to an OpenFile, which points to a Buffer, which points back to the BufferedFile.

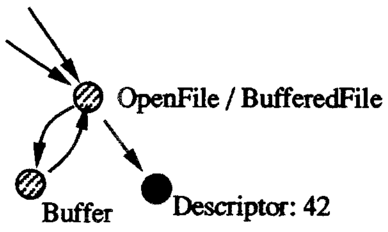

Fig. 6. Cyclic Finalization

A.3 Courier Handles

This example deals with "resurrection," when the finalization code for an object makes the object accessible rather than letting it become garbage. In this example, objects that are difficult to recreate are kept in a cache. Rather than being deallocated, they are recycled.

In D-Cedar, handles for open Courier connections were serially reusable: a client could use a connection for awhile, then give it back to the Courier package which might eventually hand it to some other client. This was useful because opening a connection is a heavyweight operation, so clients talking in rapid succession to the same machine got a performance boost by reusing an already-established connection. To return connections to the pool when clients dropped them, finalization was enabled for each CourierHandle. The Courier package gave out a handle to a single client. When the client finished, it either gave the handle back or dropped it. In the latter case, finalizing the object provided the missing giveback call.

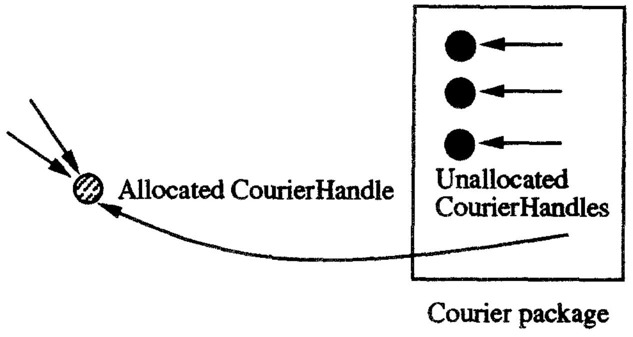

Fig. 7. Caching Finalizable Objects

A.4 Log Files

Examples of this kind were not mentioned in the main body of the article, but constitute another interesting problem with finalization. In this example, two different finalization actions need to be attached to a single object, since it needs to be finalized at two levels of abstraction.

Given an OpenFile abstraction, we might want to build a LogFile — a file to which clients could log records text records. When a LogFile is no longer accessible, we would like to write one last record to it saying that the log file has now been closed. The finalization for the LogFile, which writes the record, must be run before the finalization for the OpenFile, which closes the file.

A.5 Population of Finalizable Objects

This is another example where multiple finalizers exist for the same object, one of which may resurrect the object. The order of finalization is important, but not apparent to the system.

We may want to build a cache of OpenFiles, to ensure that two clients requesting the same file share the same OpenFile. If the cache's finalizer is run first, it cannot know when the file's finalizer has finished, and risks having two OpenFiles for the same file — one newly opened and one not quite yet finalized. If on the other hand the OpenFile's finalizer is run first, it cannot know if any pointers to the file remain, and a call might occur after finalization.

References

[AAB$^+$91] H. Abelson, N. I. Adams IV, D. H. Bartley, G. Brooks, Dybvig R. K., D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steel Jr., G. J. Sussman, and M. Wand.

Revised^4 report on the algorithmic language Scheme. *ACM LISP Pointers*, IV(3), November 1991.

[ADH$^+$89] R. Atkinson, Alan Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser. Experiences creating a portable Cedar. *SIGPLAN Notices*, 24(7):261–269, July 1989.

[AN88] Martin C. Atkins and Lee R. Nackman. The active deallocation of objecs in object-oriented systems. *Software Practice and Experience*, 18(11):1073–1089, November 1988.

[AP87] S. G. Abraham and J. H. Patel. Parallel garbage collection on a virtual memory system. In *14th Annual international symposium on computer architecture*, page 26, June 1987.

[Atk89] Martin C. Atkins. *Implementation Techniques for Object-Oriented Systems*. PhD thesis, University of York, Dept. Computer Science, University of York, Heslington, York, YO1 5DD, England., 1989.

[Bar89] Joel F. Bartlett. Mostly-copying garbage collection picks up generations and C++. Technical report, Digital Western Reseaerch Laboratory, October 1989.

[Boe91] Hans-J. Boehm. Simple gc-safe compilation. In *OOPSLA Workshop on Garbage Collection in Object-Oriented Systems, 1992*, October 1991. Available by anonymous ftp from cs.utexas.edu in pub/garbage/GC91.

[BW88] Hans-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[Cha87] David R. Chase. *Garbage Collection and Other Optimizations*. PhD thesis, Rice University, November 1987.

[DoD91a] Department of Defense. *Mapping Rational*, volume I of *Ada 9X Mapping*. Intermetrics, Inc., Cambridge, Massachusetts, August 1991.

[DoD91b] Department of Defense. *Mapping Specification*, volume II of *Ada 9X Mapping*. Intermetrics, Inc., Cambridge, Massachusetts, August 1991.

[Det91] David L. Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie-Mellon University, 1991.

[Ede90] D. Edelson. Dynamic storage reclamation in C++. Technical Report UCSC-CRL-90-19, University of California at Santa Cruz, June 1990.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, Reading, Mass, 1990.

[Hud91] Richard L. Hudson. Finalization in a garbage collected world. In *OOPSLA Workshop on Garbage Collection in Object-Oriented Systems, 1992*, October 1991. Available by anonymous ftp from cs.utexas.edu in pub/garbage/GC91.

[LHL$^+$77] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language euclid. *SIGPLAN Notices*, February 1977.

[NLT$^+$86] Lee R. Nackman, Mark A Lavin, Russell H. Taylor, Walter C. Dietrich, Jr., and David D. Grossman. AML/X: a programming language for design and manufacturing. In *Proceedings of the Fall Joint Computer Conference*, pages 145–159, November 1986.

[Nyb89] Karl A. Nyberg, editor. *The Annotated Ada Reference Manual*. Grebyn Corporation, Vienna, Virginia, 1989. [An annotated version of ANSI/MIL-STD-1815A-1983, The Ada Reference Manual].

[Par90] ParcPlace Systems. *ObjectWorks / Smalltalk User's Guide, Release 4*. ParcPlace Systems, Inc, Mountain View, CA, 1990.

[RAM84] Jonathan A. Rees, Norman I. Adams, and James R. Meechan. The T manual. Technical report, Yale University, January 1984.

[Rov85] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, staticly-checked, concurrent language. Technical Report CSL-84-7, Xerox

Corporation, July 1985.

[SMS81] Richard L. Schwartz and P. M. Melliar-Smith. The finalization operation for abstract types. In *Proceedings of the 5th International Conference on Software Engineering*, pages 273–282, San Diego, California, March 1981.

[Sun90] Sun Microsystems. *NeWS 2.1 Programmer's Guide.* Sun Microsystems, Inc, Mountain View, CA, 1990.

[Xer85] Xerox Corporation. *Interlist Reference Manual*, volume 1. Xerox Corporation, Palo Alto, CA, October 1985.