Ephemerons: A New Finalization Mechanism

Barry Hayes[1]

1.BHayes@placeware.com, 2037 Landings Drive, Mountain View, CA 94043 Phone 415 944 0900, Fax 415 944 0929

ABSTRACT

Finalization occurs when a garbage collector informs an application that an object is "almost collectable." It is used to help an application maintain its invariants. To make finalization more useful, this paper defines "almost collectable" in terms of a new class of objects, called *ephemerons*. Ephemerons are similar to weak pairs, but an object in an ephemeron's key field may be classed as "almost collectable" even if it is reachable from the epehemeron's value fields.

Keywords

garbage collection, finalization, weak pointers, resource management

INTRODUCTION

Languages with garbage collection have had ardent followers since early Lisp systems, and forms of *weak references* have been around since at least the early 1980's [RAM84, Xer85]. Weak references allow a collector to free certain objects, even when there are some bookkeeping pointers to them, as will occur with object caching or property associations.

Objects requiring type-specific clean-up activity at deallocation benefit when *finalization* is added to the collector. Finalization just requires that the collector notify the application when designated objects are collected [Rov85, Par90].

Road map

In this paper, we first discuss garbage collection and finalization in more detail, with a focus on using finalization to manage external resources. Using a certain class of finalization, involving *non-resurrecting* collectors, we show a problem case where objects needing finalization are being inappropriately retained by pointers contained in information needed for their own finalization.

A variant of this problem, independent of resurrection, is then demonstrated in implementations of *property tables* using weak references. Simply associating an object with a property is enough to keep it from being collected, based only on the external semantics of property tables. Both of these problems are known in the garbage collection community, but do not seem to have been discussed in the literature.

Finally, we introduce *ephemerons* as a variant on weak pairs, a traditional structure using weak references, and show how the garbage collector can trace ephemerons to solve both of the problems presented. A pseudo-code implementation of an ephemeron-aware garbage collector is presented at the end of this paper.

For a comprehensive overview of uniprocessor garbage collection and automatic memory management, see Paul Wilson's excellent survey article [Wil92]. For an overview of finalization, see [Hay92].

Clarification

The author of this paper is just that, and is not the inventor of ephemerons. George Bosworth invented ephemerons and designed the algorithms presented in this paper.

GARBAGE COLLECTION

Dynamically garbage collected languages avoid many common errors that occur when deallocation must be explicit. There are situations where explicit deallocation might be more efficient, clearer, or easier, but garbage collection is becoming ever more common.

The concepts presented in this paper could be implemented in a reference counting collector, but not with great ease. All further discussion will assume that the garbage collector is some variant of a tracing collector, perhaps generational [FY69, LH83], and conservative [BW88].

In one view, the garbage collector's purpose is to supply information of a global nature concerning how objects are connected to one another. The simple case is that it is being asked to locate and deallocate those objects that have become unreachable from some specified set of root objects.

Given the view of the collector as the source of topological information, it's natural to ask what other services can be driven by this supply of information. As with simple garbage collection, the information could be gathered by other mechanisms, such as reference counting [Col60, DB76]. This discussion will assume that the collector is the only source of the topology, and the goal is to find other services that require the same sort of view of the global topology as the garbage collector. In other words, we are looking for problems that could be addressed by a solution like reference counting, and trying to see if the garbage collector's traversal of memory can be used instead.

FINALIZATION

While garbage collection can handle the deallocation of unneeded objects, there are situations where a little more help from the garbage collector makes life much easier for the designer of an application.

The garbage collector can be augmented to provide information to the application, through a *finalization interface*, to let the application know when an object of interest is *"almost collectable."* The precise definition of the finalization interface is directly linked to the definition of "almost collectable."[1]

One canonical example where finalization is useful is in handling objects that are proxies for externally allocated resources, like files supplied by an operating system. In this case, the operating system has a protocol it expects users of the file to follow: a file is opened, it is read and written, and then it is closed.

If the garbage collector simply discards the proxy object that represents this system resource, the protocol isn't strictly followed because the proxy might be discarded before the file is closed. This may lead to unflushed buffers, files that are unavailable to other processes, and may even cause the operating system to run out of resources associated with open files.

If the proxy object were somehow special, the garbage collector could be modified to notice when a file was about to be collected and close the file before the proxy's memory is deallocated. Some Lisp systems have explicitly allowed files to have this behavior [AAB+91, RAM84], but without generally making it available for other application objects.

The mechanisms in this paper are a variant of *container-based finalization.*[2] In these schemes, the proxy objects are seen as special not because of any feature of their own, but because they are in some special kind of container recognized by the garbage collector.

For example, to make sure that all files are closed when the application is done with them, the file manager could maintain a special "open-file container," known to the collector and the file manager, but not to the file manager's clients. The file manager can put every file proxy into that container when it is first opened, and pass a pointer to the proxy back to the client opening the file.

As long as there is a client actively using the file, there will be a pointer to the proxy other than the pointer in the open-file container. When the client deletes the last pointer to the proxy there will still be a pointer in the file manager's open-file container. If the manager knew that the proxy was "almost collectable," by this definition, it could know to close the file and free the resource, and could then collect the proxy.

This is precisely the kind of "global topology" information that the garbage collector is in a position to discover and communicate.

THE PROBLEM

There are two actions that are indicated for an object that is "almost collectable:" one is to run the specific code to free the system resource, the other is to free the proxy. Some collectors take the view that proxies should be collected first, before the application is notified. The pointers to the proxy from the special containers are replaced by some special *tombstone* (or perhaps a null pointer) to avoid dangling pointers to the deallocated proxy. Once the tombstones are in place, there are no pointers to the object and it can be deallocated. Only then will the application be notified. This avoids having the "almost collectable" proxies *resurrected*, since the "almost collectable" objects are collected before the application is notified.[3]

In the open-file example, the proxy would be freed by a non-resurrecting garbage collector before the file manager would be notified that the proxy was "almost collectable." This means that information needed to close the file (the file ID number, for example) must be separate from the proxy itself, since the proxy will have been deallocated by the time the file manager is notified that a file proxy was collected. The information needed throughout the lifetime of the resource is factored into the proxy and the *executor*, with the executor holding the information that will be needed after the proxy has been collected.

Proxies are known to the garbage collector, and are the internal object representing some external resource. Executors are not known to the garbage collector, but are a conventional name used for an object that holds information about an external resource that is needed to finalize the external resource after the proxy has been garbage collected.

In most cases the factoring is simple, and a shallow copy of the object can be used as its own executor. But if the object

1. In modern collectors, not all unreachable and "almost collectable" objects will be detected by each collection. This can happen due to the presence of *conservative pointers*, data values which must be assumed to be pointers, but might not be, as well as generational collection, where not all objects are targeted for collection at all times.
These collectors may assume that certain objects are reachable when they are not, and unreachable and "almost collectable" objects may be treated as live by the collector. Finalization is not prompt in that a finalization action may be triggered long after the event that made the object "almost collectable."

2. As opposed to *object-based finalization*. In a simple form of object-based finalization, seen in Java, the collector arranges that proxies receive a special message when they become unreachable from the roots [GJS96]. This also typically makes them reachable again.

3. A non-resurrecting collector is supposed to make it easier to form and check some invariants, but this opinion is not universal.

contains a pointer to itself the executor ends up with a pointer to the proxy, and as long as the executor is reachable the proxy will never be "almost collectable." The path from the executor to the proxy need not be short -- any indirect path that leads from the proxy to itself will induce a path from the executor to the proxy, and thwart finalization.

The factoring of proxies and executors can get complex. If there are two finalizable objects, each may prevent the other from being finalized by referring to the other's proxy in its executor. The factoring to avoid this may require more global knowledge that can easily be expressed in interfaces, since an interface typically does not indicate that an object is a proxy for an external resource that will require finalization Loops of finalizable objects present difficult problems outside the scope of this paper.[1]

Only non-resurrecting collectors force the issue of proxy/ executor factoring. In a collector that does not tombstone the pointers from the special containers, the proxy and executor can be collapsed.

The problems associated with factoring into proxies and executors is associated with non-resurrecting collectors. There is a more general problem involving circular references to "almost collectable" objects that is present in any collector supporting container-based finalization: freeing of objects from property tables.

A BASIC WEAK CONTAINER

Container-based finalization requires .special constructs known to the collector in order to define "almost collectable." One early and elegant solution was to define "almost collectable" in terms of a new construct called *weak pointers*[2] [RAM84, Xer85, ADH+89, Par90].

A weak pointer can be traversed just like any other pointer, but is treated specially by the collector. When a collector supporting finalization traces to find all reachable objects, it traces in two phases, rather than one. The first phase does not trace through weak pointers, but queues them for later processing. The second phase starts at the enqueued pointers and traces through all pointers. The objects that are not reached in the first phase, but are reached in the second are "almost collectable." These are just those objects that can be reached from the roots, but would be unreachable if all the weak pointers were replaced by tombstones.

As a concrete example, consider implementing a *weak pair* with a special Lisp *cons cell* where the *car* (the first element of the pair) is weak but the *cdr* (the second element) is not, as in Figure 1. The double box is a cons cell, and the first field, the *car*, is shaded to show that it contains a weak pointer. If the weak pair in the figure is reached in the first tracing phase, the tracing will continue along the *cdr* of the pair, the second field, to the unshaded circle, but the *car* will not be traced. In the second tracing phase, if the object represented by the black circle has not been reached, that object is "almost collectable." The collector will notify the application, and then tombstone the pointer or trace into and through the object represented by the black circle.

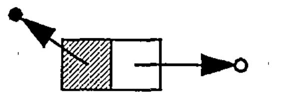

Figure 1
A weak pair

The notification to the application may take many forms; the method chosen is of no consequence in this paper.[3] But the notification must be exact enough that an application can determine which objects are "almost collectable." For concreteness, we will assume that the collector sends a message to a weak pair when it detects that its *car* is not reachable except via this link and other *cars* of weak pairs, and we will also assume that the method invoked by that message can be overridden to provide the particular behavior needed by any particular weak pair.

Weak pairs can be used to build an open-file container that allows management of proxies. In Figure 2, a group of weak pairs is acting to create a *weak collection.* File proxies (represented by the black circles) are chained on to this list when they are opened by the file manager. When the last client pointer to a proxy is dropped, the collector will send a message to the weak pair, which can close the file, and remove the weak pair from the collection.

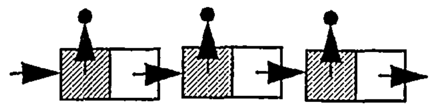

Figure 2
A weak collection

1.When a group of finalizable objects that point to one another are all are found to be "almost collectable" there is no general agreement as to which, if any, should be informed. The difficulty is that one may need another at finalization time, and if an inappropriate order is chosen, it may find that the needed object has already been finalized. When the connectivity among the objects contains cycles, the collector needs extra information to make a reasonable choice. *Guardians* [DBE93] are one attempt to supply that information, but this approach has not been widely accepted.

2.Weak pointers have gone under many other names, including *xpointers, soft pointers, and hashed pointers.*

3.Implementations differ in how they notify the application to identify the set of "almost collectable" objects. The notification must contain enough information to identify a particular weak slot. For a weak pair, having just one slot, the identity of the pair would suffice. For a structure having many weak slots, an object/index pair would be needed.

PROPERTY TABLES

Weak pairs can also be used to construct tables of key/value pairs to add arbitrary properties to arbitrary objects. The added benefit of using weak pointers to build these tables is that in most cases adding a property to an object does not change the time when it is garbage collected. When the property tables act in concert with the collector, the added property can work more or less as an added instance variable as far as reasoning about objects' lifetimes.

For a concrete implementation, we could have a property table use two cons cells per entry, one containing a key in a weak slot and the other containing a value in a strong slot, as in Figure 3. When a queried object is found in the "key" position, the probe returns the value from the associated "value" cell. This will make a perfectly respectable property table.

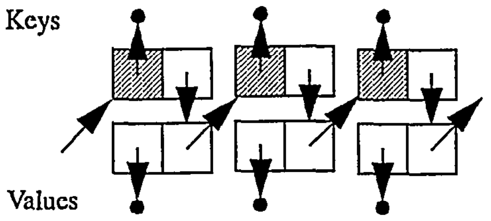

Figure 3
A property table

If all of the pointers in the property table were strong pointers, then simply entering a key/value pair in the table would ensure that they would never be garbage collected without extra effort. When the keys are held by weak pairs, finalization lets the table purge itself of keys and values that are no longer useful.[1]

If the only pointers to some key are from the weak pairs of property tables, the garbage collector will not find the key in phase one, but will find it in phase two. The weak pair will be notified that the key is "almost collectable" and can arrange that the key and value be removed from the table. The key is released when the garbage collector discovers that no client external to the tables will be able to get the key, and thus no query on the tables with that key is possible.

A problem

The problem comes about when the "values" slot of some entry contains a direct or indirect reference to a key. An object might have itself as the property value, as do the first two cells in Figure 4. The implementation of a property with a value still works -- when the object is queried in the property table, it returns itself as its value. But when all other references to the object are deleted, the reference from the "value" slot lets the garbage collector find the key in phase one of the trace. The definition of "almost collectable" is not quite what is wanted, since the presence of an object in a value slot makes the object uncollectable.

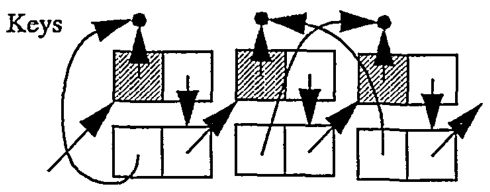

Figure 4
Some problems with weak pairs

Making the "value" slots weak doesn't help either. If some object only exists as a value in a property table, the table must keep that entry, since the key that maps to that value may not have been collected, and so a query on that key is still possible.

Other Weak Variants

Weak pairs and this implementation of property tables were chosen as a simple way to get weak pointers and notification into the system, and there are many variants we could have chosen instead. Perhaps some other simple variant of weakness would not have this problem.

Unfortunately, the problem is inherent in two-phase tracing and the definition of "almost collectable" that it imposes. Without considering the implementation of property tables, we can discuss their characteristics. For example, given a property table and a key, there must be a "get value" operation to recover the key's value, if it has one.

Likewise, the relationship to garbage collection can be well defined. We would like a key/value pair to be collected when no series of "get value" operations starting with an object reachable without the tables could reach the key. In these situations, no application can reach the key, even though it is in a table.

But how can a two-phase collector trace property tables to construct proofs that a key/value pair should be collected? Consider an object that has itself as its own property value, when there are no other pointers to the object. This object should be "almost collectable." If the table's values are traced in the first phase, then this object would not be "almost collectable," as desired.

If the table's values are not traced in the first phase, consider an object which is a value in a property table when there are no other pointers to it. This object should not be "almost collectable" but would not be traced in the first phase.

The definition of "almost collectable" that two-phase tracing generates doesn't jibe with how property tables should be garbage collected.

EPHEMERONS

The case where a property table contains the same object as its key and value fields, and the object is not stored anywhere else, is one example of an *unreachable property*. This is an abstract quality of properties independent of implementation.

1.This presupposes what it means to be "useful." In particular, we assume that there is no way to ask the table for the keys. So, for example, there is no "which keys have this value" operator.

Another example is two properties, each with the other's key as its value, as in the second two properties in Figure 5. In general, if a property's key is only reachable through unreachable properties, it is an unreachable property.

An implementation of property tables would like to know it may free an association when the property it represents is unreachable. "Almost reachable" needs to be refined to take these unreachable properties into account.

Ephemerons are a refinement of weak pairs that solve this "unreachable property" problem. The first slot of an ephemeron is used to hold the key of a property, while the second slot holds the value, but the slots are neither "weak" nor "strong" in the previous sense.

Tracing Ephemerons in the Garbage Collector

An ephemeron-aware collector traces objects in three phases rather than two. The first phase traces up to ephemerons, but traces none of their fields. The second phase repeatedly traces ephemerons that can be classed as not maintaining unreachable properties. At the end of the second phase, the remaining ephemerons all represent unreachable properties. The third phase traces all remaining reachable objects (or tombstones the pointers to them).

Ephemerons are classed by the garbage collector as either maintaining a "reachable property" or not. When an ephemeron is encountered in the course of the first phase of a garbage collector trace, the collector does not immediately trace *either field* in the ephemeron, but puts it on a queue. Ephemerons in this queue might maintain reachable or unreachable properties.

In the second phase, the collector scans the queue of delayed ephemerons. Any ephemeron that has a key that has already been reached maintains a property that could be requested -- the ephemeron is reachable, and the key is reachable, so some code might ask for the value. These are classed as "reachable properties."

Any ephemeron that has a key that has not been reached may or may not maintain a reachable property -- that remains to be seen. These ephemerons are requeued for future inspection.

The first group of ephemerons, those maintaining reachable properties, are now traced in the same way that any non-ephemeron would be. Since the key is known to have already been reached, it has been traced, and only the value field needs to be traced.

But tracing these value fields may make it clear that some ephemeron in the queue maintains a reachable property. Ephemerons on the queue must be inspected again to see if they now can be clearly classified as maintaining a reachable property. Also, more ephemerons may be discovered as the value fields are traced. When they are discovered, they are added to the queue.

This continues until the queue contains only ephemerons that have keys that have not yet been reached. This is a set of ephemerons that maintain unreachable properties. It is not possible to request the value field of one of these ephemerons without first having the key or value field of some other ephemeron in the set. At this point the collector arranges that all of these ephemerons in the set will be notified. If the ephemerons are maintaining a collection of property tables, deleting all of these ephemerons will release the storage associated with the unreachable properties, just as if the properties had been added as instance variables.

In the third phase, the collector traces the remaining objects, beginning at the ephemerons still on the queue. Any ephemerons encountered in this phase are treated as ordinary objects, and all fields traced[1].

The portion of the algorithm presented that traces ephemerons adds running time $O(nd)$ where n is the number of ephemerons and d is the length of the longest chain of ephemerons. In practice, d is small and the algorithm is linear in the number of ephemerons[2].

Managing External Resources with Ephemerons

Ephemerons solve a sticky problem in property tables, but surprisingly they also solve the problem in managing external resources using proxies and executors.

Recall that information needed to return a resource to its external manager may be kept in an executor rather than a proxy. When this occurs, the split between executor and proxy must be carefully managed. With only weak pairs, the executor cannot keep a strong reference to the proxy itself or any indirect path that leads to the proxy, since that would cause the collector to find the proxy in phase one of the trace.

If the executor is in a value field of an ephemeron that has the proxy as its key, it may contain references to the proxy without interfering with the finalization. The proxy in the key field guards the executor in the value field. The collector will not trace the executor unless the proxy is reachable anyway. The value fields are traced only after the key is shown to be otherwise reachable, in which case finalization should not occur, or after the collector has decided that the ephemeron maintains an unreachable property, in which case the application will be notified to finalize the resources.

Further Variants of Ephemerons

A simple extension to ephemerons is to allow flexibility in the number of "value" fields. The first field of an ephemeron is the "key," and all other fields are treated the same way as the "value" field in a simple ephemeron[3].

This extension applies even if there is only a single field. A one-field Ephemeron simply has the "key" slot, and is a one-element weak container. It can be used to mimic traditional

1.It is unclear if they should also be sent a message. Strictly speaking, they are maintaining unreachable properties, and if they had been discovered in phase one or two, because of a conservative pointer for example, they would not effect the results of the trace, but would have been included in the set. Of course, if in addition their keys had been discovered due to another conservative pointer, they would not have been so notified.

two-phase weak tracing.

Three-phase tracing and ephemerons offer a slight variation on what most container-based finalization systems already do, and so most variants of two-phase tracing can be retooled for three-phase tracing. The ephemeron's references may be tombstoned rather than traced, the notification to the application may be synchronous, and so on.

CONCLUSIONS

Ephemeron tracing, as is implemented with the three-phase algorithm described, solves some long-standing problems with finalization.

- Packages responsible for management of external resources can maintain pointers to the proxies for those resources without interfering with the collector's ability to decide when a proxy indicates a resource that needs to be returned to its external manager.
- Objects can be removed from property tables when they are no longer useful, even if they are used as values of properties of other objects.

LEGAL NOTICE

I am not a lawyer, and this section should not be taken as legal advice.

Ephemerons have been a trade secret in some Digitalk and ParcPlace-Digitalk products shipped more than one year ago, and ParcPlace-Digitalk has allowed publication of this paper, making this trade secret public knowledge. It is my understanding that because more than a year has passed since shipping those products, a valid United States patent for ephemerons would not be issued to the inventor.

I do not know if any other obstacles, such as foreign patent rights, should discourage anyone else from implementing ephemerons.

Java, VisualWorks, and VisualSmalltalk, are trademarks.

2.Ephemerons can be recovered in time strictly proportional to the number of ephemerons. To do this, build distinct delay queues for each unique key that has not yet been visited. When the trace encounters an ephemeron, if the key has already been traced, there will be no delay queue, and the ephemeron should be traced immediately. If the key has not been traced, there will be a delay queue and the ephemeron should be enqueued.
When any object which is a key for some delayed ephemerons is found to be reachable, its associated queue of delayed ephemerons should be traced at that time, and the queue deleted.
Any ephemeron that is left on any queue after this single trace phase maintains an unreachable property. These ephemerons need to be traced in a second phase.
3.This extension is allowed in VisualWorks's and VisualSmalltalk's implementation of ephemerons.

ACKNOWLEDGMENTS

Ephemerons were developed by employees of Digitalk, Inc, and this paper was written with the support of many people at ParcPlace-Digitalk, Inc.

Ephemerons were designed by George Bosworth. They were first implemented for VisualSmalltalk by Roger Thayer. Barry Hayes implemented them for VisualWorks.

REFERENCES

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

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

[COL60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655-657, December 1960.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. Communications of the ACM, 19(9):522-526, September 1976.

[DBE93] R. Kent Dybvig, Carl Bruggeman, and David Eby. Guardians in a generation-based garbage collector. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6):207-216 of ACM SIGPLAN Notices, Albuquerque, New Mexico, June 1993. ACM Press.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. Communications of the ACM, 12(11):611-612, November 1969.

[GSJ96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, Addison -Wesley, August, 1996.

[Hay92] Barry Hayes "Finalization in the Collector Interface" in *Memory Management, Proceedings of The International Workshop on Memory Management, 1992, St. Malo, France, September, 1992*, Y. Bekkers and J. Cohen, editors, LNCS 637:277-298, Springer-Verlag.

[LH83] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. Communications of the ACM, 26(6):419-29, 1983. Also report TM-184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[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, statically-checked, concurrent language. Technical Report CSL-84-7, Xerox PARC, Palo Alto, CA, July 1985.

[Wil92] Paul R. Wilson. "Uniprocessor Garbage Collection Techniques" in *Memory Management, Proceedings of The International Workshop on Memory Management, 1992, St. Malo, France, September, 1992*, Y. Bekkers and J. Cohen, editors, LNCS 637:1-42, Springer-Verlag.

[Xer85] Xerox Corporation. Interlisp Reference Manual, Volume 1. Xerox Corporation, Palo Alto, CA, October 1985.

EPHEMERON COLLECTION CODE

Send the mark message to every object that's reachable from the roots.

```
Heap::garbageCollectMark
   self markPhase1.
   self markPhase2.
   self markPhase3
```

Mark all of the objects from the roots up to any *ephemerons*. Place the ephemerons in the global collection EphemeronQueue.

```
Heap::markPhase1
   EphemeronQueue makeEmpty.
   self enumerateRoots: [:rootPointer |
      rootPointer tracePointerQueueingEphemerons]
```

Mark all of the objects reachable from this pointer, queueing all of the ephemerons encountered.

```
Pointer::tracePointerQueueingEphemerons
   self deref isMarked
      return.

   self deref markObject.
   self deref isEphemeron
      EphemeronQueue add: self.
   else
      self deref enumeratePointers: [:pointer |
         pointer tracePointerQueueingEphemerons]
```

Identify a set of ephemerons with reachable keys, and trace them. Since that might cause other ephemerons' keys to become reachable, recurse until all the ephemerons on the queue have unreachable keys.

```
Heap::markPhase2
   | reachableProperties otherProperties |

   otherProperties <- Collection new.
   reachableProperties <- Collection new.
   EphemeronQueue enumerate: [:ephemeron |
      ephemeron deref key isMarked
         reachableProperties add: ephemeron.
      else
         otherProperties add: ephemeron].
   EphemeronQueue <- otherProperties.

   reachableProperties empty not
      reachableProperties enumerate: [:reachableProperty |
         | value |
         value <- reachableProperty deref valueField.
         value tracePointerQueueingEphemerons].
      self markPhase2
```

The queue contains a collection of ephemerons that maintain unreachable properties. Notify the ephemerons, and trace through.

```
Heap::markPhase3
   EphemeronQueue enumerate: [:ephemeron |
      ephemeron deref signal: almostCollectable.
      ephemeron deref keyField tracePointer.
      ephemeron deref valueField tracePointer]
```

Mark all of the objects reachable from this pointer, paying no attention to *ephemerons.*

```
Pointer::tracePointer
   self deref isMarked
      return.

   self deref markObject.
   self deref enumeratePointers: [:pointer |
      pointer tracePoiner]
```