Extending Garbage Collection to Complex Data Structures

Laura Effinger-Dean
Williams College
Williamstown, MA 01267
laura.effinger-dean@williams.edu

Chris Erickson
Harvey Mudd College
Claremont, CA 91711
cerickso@cs.hmc.edu

Melissa O'Neill
Harvey Mudd College
Claremont, CA 91711
oneill@acm.org

> Beware lest you lose the substance by grasping at the shadow.
>
> —Aesop

ABSTRACT

Objects that are pointer reachable through a complex data structure may be inaccessible to the external program, depending on the semantics of the structure. Failure to recognize the unusual behavior of complex data structures causes memory leaks in any collector that relies on pointer reachability to locate garbage. We extend the definition of reachability to distinguish between objects that are reachable to the program at large and objects that are within the interior of structures. Our general mechanism allows any structure to run arbitrarily complex collection algorithms during normal garbage collection.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures; D.3.4 [**Programming Languages**]: Processors—*Memory management*

1. INTRODUCTION

If a running program is never going to use a particular object again, the object is *garbage*: it may safely be discarded and its memory reclaimed.[†] As the average garbage collector cannot foresee a program's actions, it is conventional to settle for *pointer reachability*, wherein the collector traces or counts the pointers to an object. Though pointer reachability is an intuitive and logical way to distinguish live objects from garbage, it falters when combined with complex data structures that use data abstraction. One example of a structure that is difficult to garbage collect is a hash map that stores key/value pairs, using pointer equality to distinguish keys. For a given pair, the key is an object A and the value an object B (Figure 1(a)). If A is not reachable from outside the map, B is inaccessible. A traditional tracing collector ignores this layer of abstraction and preserves both A and B.

In order to properly collect complex data structures, lines of communication must be established between the program and the garbage collector. Several garbage-collected languages, including Java, offer support for *weak pointers*. Weak pointers maintain a reference to an object but are ignored by the collector. In the key/value problem, making the key a weak pointer means that the collector preserves object A only if a non-weak pointer to A exists (Figure 1(b)). More intricate arrangements of pointers require, in turn, more sophisticated methods for identifying garbage. In some cases, it may be impossible to tell that an object is garbage without first interpreting the data it contains.

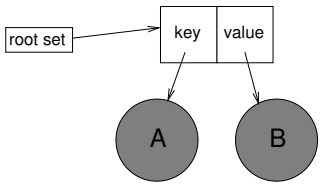
Persistent data structures [1, 6, 5, 7, 2, 3, 14, 13] are particularly hard to garbage collect. These structures simulate familiar data structures such as arrays or linked lists, using a complex web of nodes and pointers to maintain information about older versions of the structure. Ideally, when a collector determines that a version is inaccessible, it should be able to safely remove that version from the persistent structure, but traditional tracing collectors do not do so for these data structures. Specialized collection algorithms exist for some of these structures, including persistent arrays [9].

Properly collecting a persistent structure requires both a detailed understanding of its semantics (rather than merely its topology) as well as knowledge (likely from the collector) about which versions are reachable. We could hack the garbage collector to recognize these structures, but that approach gets ugly quickly if we want to use more complex structures or different garbage collectors. A better solution would be to allow programs to communicate with the garbage collector and specify collection algorithms for arbitrarily complex data structures.

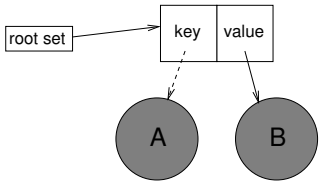
Another advantage to this customized collection scheme is a matter of ideology. Languages such as Smalltalk, ML, and Java allow programmers to harness the power of data abstraction; it seems appropriate that their garbage collectors should be able to respect abstraction where necessary and harness its power when doing so is useful. Dan Ingalls, one of the original designers of Smalltalk, criticized languages that encourage programmers to violate abstraction, describing such abuse as "plundering data structures" [11], yet garbage collectors typically disregard all boundaries around data structures and plunder away. This shortsight-

[†]The term "object" refers to an allocated piece of memory, and has nothing to do with object-oriented programming.

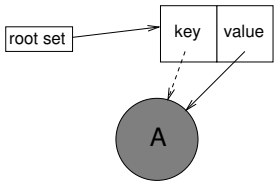
SPACE 2006 Charleston, South Carolina, USA



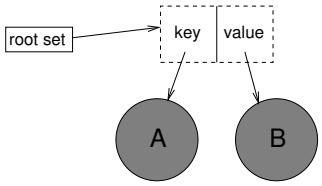
(a) A simple key/value pair.



(b) A key/value pair using a weak pointer for the key.



(c) A case where using a weak pointer fails.



(d) The dashed box is an ephemeron.

1: Garbage collecting a key/value pair.

edness leads to memory leaks in every basic tracing collector.

In this paper we develop a framework for collecting complex data structures with customized collection algorithms. Our contributions are as follows:

1. We discuss how key/value pairs or *ephemerons* [10, 12] may be extended to more complex structures (Section 2).
2. Within these structures, pointer reachability no longer implies that an object is accessible to the outside program. In Section 3 we redefine reachability in terms of the *exterior* and *interior* of each structure.
3. We present a method for implementing customized collection algorithms within a tracing garbage collector (Section 4). Our implementation takes advantage of the distinction between exterior and interior objects by performing customized collection before most of the objects on the interior of the structure have been identified.

The idea of tailoring collection to specific structures is unfamiliar and may seem bizarre, but we believe that it is an important part of data abstraction that has too long been overlooked. We hope that our ideas will encourage designers of data structures to consider exactly how their structures interact with the garbage collector.

2. KEY/VALUE PAIRS

It is helpful to start with the example we mentioned in the introduction. A key/value pair is an (admittedly simple) data structure that requires special treatment from the garbage collector. Consider the pair shown in Figure 1(a). If we use pointer equality to distinguish keys, then neither A nor B is accessible, because no outside pointer to A exists. Garbage collection using weak pointers, as in Figure 1(b), properly collects object A, but the program must manually purge object B. Also, weak pointers fail when a path to object A exists through the value, as in Figure 1(c).

2.1 Ephemerons

Hayes [10] described a structure, invented by Bosworth, called an *ephemeron* that solves the key/value collection problem. (Peyton Jones and colleagues [12] independently came up with a similar construct called a *key/value weak pointer*.) Figure 1(d) shows the use of an ephemeron. Each ephemeron contains pointers to key and value objects; the value (object B) is reachable if both the ephemeron and the key (object A) are reachable. The collector traces in three phases:

Phase I: Trace through memory, but do not trace ephemerons. Place ephemerons on a list for tracing in Phase II.

Phase II: Process the list of ephemerons as follows:

(a) Partition the list of ephemerons into a *key-reachable* list and a *key-unreachable* list.

(b) Trace the values of the ephemerons on the key-reachable list.

(c) If the key-reachable list is non-empty, repeat from step IIa using the key-unreachable list.

(d) Notify any ephemerons on the key-unreachable list.[†]

Phase III: Trace the ephemerons on the key-unreachable list normally.

Note that the collector repeatedly traverses the list of ephemerons during Phase II. This repetition is necessary because some ephemerons may contain *nested pointers* — value pointers that, when traced, lead to the keys of other ephemerons.

2.2 Extending Ephemerons

We shall extend ephemerons to more complex data structures. We call a data structure that requests customized collection a *blob*. Blobs may contain any number of objects, and may request reachability information for any number of keys. Allowing multiple keys complicates the issue of nested pointers. Say that we've determined that a certain subset of a blob's keys are reachable. We then trace the logically reachable "value" pointers and discover that one or more of the keys we'd previously classed as unreachable can be reached through the blob itself. The difficulty of identifying nested pointers means that we must gradually build up a set of reachable keys for each blob.

Collecting blobs is not as simple as deciding whether to traverse one object given reachability information about another. Rather, each blob must be able to specify an arbitrarily complex collection algorithm. We call these algorithms *cleanup functions*.

3. REDEFINING REACHABILITY

Within blobs, pointer reachability does not imply actual reachability. In other words, an object on the "interior" of a blob may be inaccessible to the program, even if there exists a path of pointers to that object. But a tracing collector does not understand the complexity of the blob, and preserves all objects—interior or exterior—that are pointer reachable. Hence it is essential not to trace pointers within the blob until after the blob has been given a chance to eliminate logically unreachable objects.

Before we get into the details of implementing specialized collection, we'll take a step back and formally define the concept of "actual reachability"—or, as we'll call it, *exterior reachability*.

An *interior* object is an object that is part of a particular blob. An *exterior* object is an object that is not interior to any blob.

An object A is *exterior reachable* if one or more of the following conditions holds:

1. It is a member of the root set.
2. An exterior object that is exterior reachable contains a pointer to A.
3. A is *logically reachable* (defined below) through an exterior-reachable blob.

[†]Hayes:97:ephemerons:oopsla does not explain what this notification accomplishes. It seems logical that the ephemerons should be deleted, but if they are part of a larger structure (e.g., a table of key/value pairs), deletion may be nontrivial. If we nullify the ephemerons' pointers, then Phase III is unnecessary. Our general mechanism shall provide a more flexible method for processing reachability information.

A blob is exterior reachable if and only if any of its interior objects are exterior reachable.

An object is *interior reachable* if it is an interior object and there is a pointer to it from another interior object from the same blob that is either exterior or interior reachable.

Intuitively, exterior reachable means that the program has access to the object, whereas interior reachable means that the object should be preserved for the blob's use.

Logically reachable objects are objects that could be accessed by the program through a blob. We can't know which objects are logically reachable without knowing the exact semantics of the blob. In practice, the collector must query the blob for a list of logically reachable objects. Logically reachable objects are, by definition, exterior reachable.

Any object whose reachability status affects the collection process for a blob is a *key object*. The collector should inform blobs of any exterior-reachable key objects. We ignore interior-reachable keys, because interior-reachable objects are not accessible to the program at large. This distinction turns out to be very helpful: we can't determine which objects are interior reachable without tracing into blobs, but we musn't trace into blobs "too early"—that is, before blobs have had a chance to perform cleanup with full reachability information. But considering only exterior reachability for keys oversimplifies matters a bit. There may be some cases where interior-reachable objects act as keys. A blob might "revive" an interior-reachable object by passing it to the program. In each of these cases, the blob containing the interior-reachable objects in question should declare them to be logically reachable.

4. IMPLEMENTING BLOBS

Given the detailed specification of blobs in Section 3, we can design an implementation without too much difficulty. The design presented herein is not the only way of implementing blobs, but rather the one that we believe naturally follows from the specification.

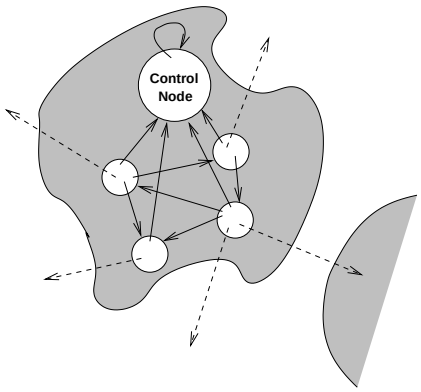
4.1 Control Nodes

Each blob has a *control node* containing a list of key objects (the objects whose reachability information affects collection) and the cleanup functions (the customized collection algorithms), further discussed in Sections 4.3 and 4.4, respectively. The control node also has a flag that is set if its blob is exterior reachable (that is, if any of the blob's interior objects are exterior reachable). Many familiar data structures already contain objects that act as control nodes. For example, the root of a tree is a control node, as is the object containing the head and tail indexes for an array-based circular queue. Each blob's control node is interior to that blob. Figure 2 shows a blob and its control node.

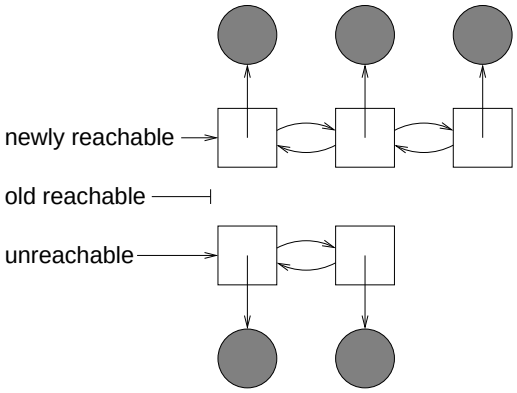
4.2 Interior Objects

We assume that it is possible to distinguish between interior and exterior objects by using a flag in the object header. Each interior object should belong to exactly one blob, in order to preserve strict boundaries between blobs. We can enforce this rule by storing a pointer to the control node in every interior object. The control node contains a pointer to itself.

We stated in Section 3 that only interior objects may be interior reachable. Though this limitation may seem obvious at first glance, consider the case in which an interior-



2: A blob, including a control node, interior objects, and weak pointers. Solid arrows are strong pointers and dashed arrows are weak pointers.



3: A key queue. Shaded circles are key objects and unshaded squares are key tags. The old reachable list happens to be empty.

reachable object points to an exterior object that is not exterior reachable. By our definition, the exterior object is not reachable, even though a pointer to it exists in a reachable object! The idea behind this unintuitive definition is that we cannot locate logically reachable objects without the blob's assistance, but we do not want the blob to hide any of these logically reachable objects. Thus pointers to a blob's exterior are ignored unless the blob has explicitly declared its contents to be logically reachable. Pointers from interior objects to objects outside the blob (either exterior or within the interior of another blob) must be implicitly *weak*. Weak pointers, as explained in Section 1, do not protect their referents from garbage collection.

In Figure 2, all of the objects on the interior of the blob contain pointers to the control node, including the control node itself. Note that all pointers extending outside the blob are weak.

4.3 Key Queues

Any object whose reachability status might affect a blob's collection process is a *key object*. These key objects may be interior or exterior, and a single object may be a key for multiple blobs. We require blobs to explicitly identify their keys in order to place a reasonable bound on collection time; if we didn't know which objects were keys, we would have to reprocess every blob if even a single new object were found during a trace.

A collection of key objects associated with a blob is called a *key queue*. A key queue consists of a number of key tags, each of which contains a pointer to a key object.

During garbage collection, the key tags are organized into three doubly linked lists as follows:

Newly reachable Key tags whose keys were discovered during the most recent trace in this garbage-collection cycle.

Old reachable Key tags whose keys were discovered during an older trace in this garbage-collection cycle.

Unreachable Key tags whose keys have not yet been discovered in this garbage-collection cycle.

We need to separate the reachable keys into two lists because we will frequently have to trace newly located, logically reachable objects. The newly reachable lists track the keys that are found during each of these traces.

After every trace, we merge the (now out-of-date) newly reachable list with the old reachable list. We then check every key tag on the unreachable list, moving any tags with reachable keys to the newly reachable list. Repeatedly traversing the unreachable lists of every key queue can be very time-consuming in the worst case. In Section 4.6 we'll present several improvements to key queues, including a way of notifying key queues in order to reduce those costly traversals; for now, we'll complete our implementation of blobs using these naïve key queues.

Key queues (see Figure 3) are very similar to *Dybvig-et-al:93:grdns-gengc:pldi* [8] or Java's reference queues [15]. Those structures inform the program when an object is *un*reachable; key queues inform a blob when an object is reachable.

4.4 Cleanup Functions

Each blob provides two cleanup functions:

getLogicallyReachable Return a list of logically reachable objects that the collector should trace. This function may be called multiple times if the collector locates previously unreachable keys.

doCleanup All exterior-reachable objects have been identified. Perform any final cleanup (e.g., redirecting pointers around unreachable objects).

getLogicallyReachable locates nested pointers, whereas doCleanup performs the blob's specialized collection algorithm. Note that doCleanup does not explicitly trace or deallocate objects; rather, the blob rearranges itself so that the collector may safely trace the interior.

Running user-supplied code during garbage collection is a quick route to data corruption, and thorough precautions must be taken. We address the design of safe cleanup functions in Section 4.7.

4.5 The Collection Algorithm

Garbage collection proceeds in four phases:

Phase I: Trace through memory, but do not trace interior objects. When an interior object is found,

(a) Place the object on a list for tracing in Phase IV.

(b) If the object's blob has not yet been flagged exterior reachable in the control node, set the flag and add the control node to a list for processing in Phases II and III.

Phase II: Locate logically reachable objects as follows:

(a) For each control node on the list,

i. Check through the key queue for reachable keys.

ii. If the key queue's newly reachable list contains any keys, or if this blob was just reached and we haven't yet called getLogicallyReachable on it, call getLogicallyReachable on the blob.

iii. Trace any logically reachable objects found, processing interior objects as we did in Phase I.

(b) Repeat step IIa until it is no longer necessary to call getLogicallyReachable on any blobs in step II(a)ii.

At the end of Phase II, all exterior-reachable objects have been identified.

Phase III: Call doCleanup for each blob.

Phase IV: Identify interior-reachable objects by tracing the interior objects on the list accumulated during Phases I and II. Nullify any dangling weak pointers.

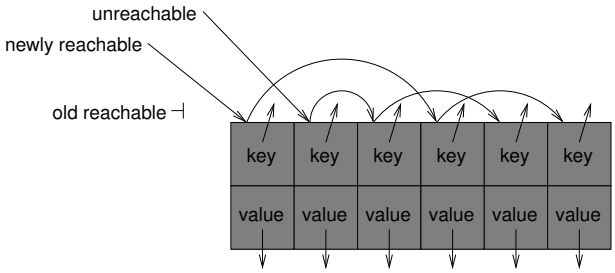
Objects traced by the collector during Phases I and II are exterior reachable; those traced during Phase IV are interior reachable. All others are garbage.

4.6 Improving Key Queues

In Section 4.3 we presented a naïve implementation of key queues that uses three doubly linked lists of key tags. The problem with this implementation is that traversing through each key queue's unreachable list may be very time-consuming in the worst case. If n is the total number of key tags over all key queues, we may have to perform up to n traces of logically reachable objects during Phase II, with each of these traces requiring retraversal of all of the unreachable lists. Thus our simple implementation of key queues has a worst-case time of $O(n^2)$.

We could avoid traversing the unreachable lists by putting a pointer to each key's key tag in the key object for each key tag and having the collector alert the appropriate key queue when the object is reached. Unfortunately, this apparently simple improvement turns out to be rather complex. First, the key object may be of any type, so we can't store a pointer to the key tag without adding an extra word to every object header. Second, an object may be the key object for an arbitrary number of key tags, so a single pointer would not suffice.

One possibility is to store information about an object's key queue in a separate structure. For example, we could use a hash table to associate the address of every key object with a pointer to its key tag. The garbage collector needs to distinguish between keys and nonkeys, perhaps using a flag in the object header. During tracing, if we reach a key object, we search through the table and move each associated key tag to its key queue's newly reachable list. If we also keep track of all blobs whose keys we found, we won't have to check every key queue for updates in step IIa of the collection algorithm.



4: An array of extended key tags within a blob.

Using a hash table to store key information means that we do constant expected work for every key tag, yielding $O(n)$ expected processing time for n key tags. Implementors averse to expected time performance can use an ordered structure and still process key queues in $O(n \log n)$ time.

Our design for key queues is still not as flexible as it should be. Imagine trying to implement a hash table as a blob using key queues and key tags. There is a one-to-one correspondence between entries in the table and key tags; if a key tag's key is exterior reachable, the corresponding value for that entry is logically reachable. Whenever an entry is added, removed, or changed, we'll have to update the key queue accordingly. Each update takes constant time, but requiring programmers to perform these updates manually is inconvenient and error-prone. In addition, the getLogicallyReachable function would be unnecessarily complicated—given a reachable key, we must search through the entire table just to find the corresponding value.

We can solve both of these problems by integrating the key tags themselves into the blob's structure. In our example of a hash table, we eliminate the old table entry and use the key tag in its place, adding an extra "value" pointer to each key tag. In languages that support inheritance, adding an extra field is easy: we simply extend the key-tag type! This change is invisible to the key queue, which carries on maintaining the reachability lists behind the scenes. We will still need to notify the key queue when removing key tags from the table, but adding a key tag or changing the key pointer will be automatic. These *extended key tags* are quite flexible and ease the process of implementing complex structures as blobs.

Figure 4 shows a blob that is using an array of extended key tags to map keys to values. Every element of the array is part of the blob's key queue. Inverse pointers for the doubly linked lists are omitted.

4.7 Ensuring Safe Cleanup Functions

As we'll be running user code during garbage collection, precautions must be taken to ensure that no data is corrupted or lost. We shall harness the tools of concurrent garbage collection to determine exactly what these precautions should be. We are not saying that blobs require the use of a concurrent collector, but rather that an appreciation of the key concepts is helpful in understanding how to run safe cleanup functions.

Concurrent garbage collection occurs while the program, or *mutator*, is running. *Tricolor marking* [16] measures the progress of the garbage-collection cycle. Each object in the system has one of three colors, depending on where the object is in the collection cycle:

Black objects have been reached and all of their pointers traced.

Grey objects have been reached, but not all of their pointers have been traced.

White objects have not been reached.

In our algorithm, we call user code at very specific points in the collection cycle—when a trace of exterior-reachable objects has completed. At these critical points, the three colors correspond exactly to our division of exterior- and interior-reachable objects:

- Exterior-reachable exterior objects are "black".
- Exterior-reachable interior objects are "grey".
- Interior-reachable objects, logically reachable objects that have not yet been identified, and garbage objects are "white".

A concurrent collector may accidentally collect nongarbage if the mutator disturbs the grey "fringe" between black and white objects. If the only path to a white object is through a black object, the collector never reaches the white object and it is discarded erroneously. However, in order to trigger this issue, the mutator must install a pointer to a white object in a black object. Translating this condition into our blob terminology, the collector could miss an object if a cleanup function installs a pointer into an exterior-reachable exterior object. Therefore, we can prevent data corruption by disallowing changes to exterior objects.

This precaution—preventing changes to exterior objects—is not unreasonable. Specialized collection applies only to the blob itself, and any changes to the layout of memory should ideally be contained to the blob's interior.

5. RELATED WORK

The difficulty of garbage collecting complex structures has seen little discussion in the literature. Detlefs and Kalsow [4] discuss the failure of garbage collectors to recognize data abstraction, and present profilers for Modula-3 designed to catch memory leaks. We believe that ours is the first attempt to design a means by which structures can provide their own specialized collection algorithms.

Java's reference queues [15] and Dybvig's Dybvig-et-al:93:grdns-gengc:pldi [8] allow programmers to process objects that are "almost dead." Our key queues similarly track the reachability of certain objects, but the introduction of logically reachable objects makes it much more difficult to determine that an object is almost dead.

Ephemerons and their Haskell counterpart, key/value weak pointers, are an elegant solution to the key/value collection problem [10, 12] and were the inspiration for many of our design decisions. Interestingly, both ephemerons and key/value weak pointers suffer from the same $O(n^2)$ worst-case bound as our naïve key queues from Section 4.3; the improvements to key queues in Section 4.6 should apply to key/value collection as well.

Finalizers, especially in combination with weak pointers, are sometimes suggested as way to make garbage collection handle difficult data structures, because arbitrary code can be executed when a particular object is collected, but they do not not solve the problems addressed by ephemerons, nor the generalization of ephemerons we have presented here.

6. CONCLUSION

We have redefined reachability to distinguish between the interior and exterior of complex data structures. Taking advantage of our definition of exterior-reachability, we have shown how to construct structures as blobs with customized cleanup functions, and designed a collection algorithm that locates all exterior- and interior-reachable objects while safely running each cleanup function.

We hope that our work, in addition to presenting a viable extension to any tracing garbage collector, will alert programmers to the difficulties incurred by combining data abstraction with garbage collection. As we have seen, it is easy to create a data structure that conceals subtle memory leaks. Persistent data structures, for example, may automatically preserve all changes to a structure, even if some versions are inaccessible to the running program. The resulting increase in memory usage is unacceptable in long-lived server applications.

We designed blobs with a single goal in mind: to allow any data structure to specify its own arbitrarily complex collection algorithm. Perhaps our approach seems odd: why a "one size fits all" solution, instead of a set of smaller tools? After all, we have no guarantee that every structure will fit perfectly into our design of blobs. The answer is that we have not dismissed the idea of small tools entirely; on the contrary, it is straightforward to combine the blobs with, say, ephemerons. We believe, however, that user-supplied code is a powerful and relatively easy to understand tool for collecting structures. Blobs are the simplest method we have for implementing these customized collection algorithms.

7. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant CNS-0451293 to Harvey Mudd College.

8. ADDITIONAL AUTHORS

Additional authors: Darren Strash (CSU Pomona, Pomona, CA 91768), email: `djstrash@csupomona.edu`)

9. REFERENCES

[1] A. Aasa, S. Holmström, and C. Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3):490–503, 1988.

[2] T. Chuang. Fully persistent arrays for efficient incremental updates and voluminous reads. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 110–129, Rennes, France, 26–28 Feb. 1992. Springer Verlag.

[3] T.-R. Chuang. A randomized implementation of multiple functional arrays. In *LFP '94: Proceedings of*

the 1994 ACM conference on LISP and Functional Programming, pages 173–184, New York, NY, USA, 1994. ACM Press.

[4] D. L. Detlefs and B. Kalsow. Debugging storage management problems in garbage-collected environments. In *USENIX Conference on Object-Oriented Technologies*. USENIX Association, 1995.

[5] P. F. Dietz. Fully persistent arrays (extended abstract). In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Proceedings of the Workshop on Algorithms and Data Structures: WADS '89*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74, Berlin, Aug. 1989. Springer Verlag.

[6] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[7] J. R. Driscoll, D. D. K. Sleator, and R. E. Tarjan. Fully persistent lists with catenation. *Journal of the ACM*, 41(5):943–959, Sept. 1994.

[8] R. K. Dybvig, C. Bruggeman, and D. Eby. Guardians in a generation-based garbage collector. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 207–216, New York, NY, USA, 1993. ACM Press.

[9] L. Effinger-Dean, C. Erickson, M. O'Neill, and D. Strash. Garbage collection for trailer arrays. In *SPACE 2006*, Charleston, South Carolina, USA, 14 Jan. 2006.

[10] B. Hayes. Ephemerons: A new finalization mechanism. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 176–183, New York, NY, USA, 1997. ACM Press.

[11] D. H. H. Ingalls. Design principles behind smalltalk. *BYTE Magazine*, Aug. 1981.

[12] S. L. P. Jones, S. Marlow, and C. Elliott. Stretching the storage manager: Weak pointers and stable names in haskell. In *IFL '99: Selected Papers from the 11th International Workshop on Implementation of Functional Languages*, pages 37–58, London, UK, 2000. Springer-Verlag.

[13] H. Kaplan, C. Okasaki, and R. E. Tarjan. Simple confluently persistent catenable lists. *SIAM Journal on Computing*, 30(3):965–977, June 2001.

[14] M. E. O'Neill and F. W. Burton. A new method for functional arrays. *Journal of Functional Programming*, 7(5):487–514, Sept. 1997.

[15] Sun Microsystems. *Java 2 Platform, Standard Edition, v 1.4.2: API Specification*, 2003.

[16] P. R. Wilson. Uniprocessor garbage collection techniques. In *IWMM '92: Proceedings of the International Workshop on Memory Management*, pages 1–42, London, UK, 1992. Springer-Verlag.