Garbage Collection for Trailer Arrays

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

ABSTRACT

Persistent data structures often use tricks to achieve their performance guarantees. In these structures, pointer reachability fails as a method of determining what is and isn't garbage. We present an extension for a garbage collector which can understand and collect a particular persistent structure called trailer arrays. Our method does not increase the time or space complexity of garbage collection in most cases.

Categories and Subject Descriptors

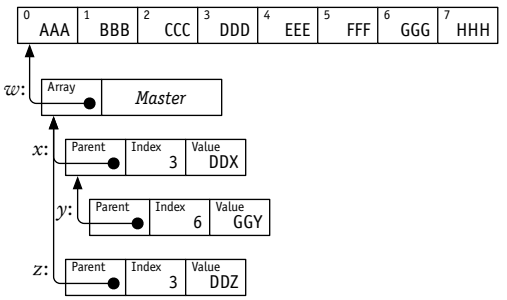
E.1 [**Data**]: Data Structures; D.3.4 [**Programming Languages**]: Processors—*Memory management*

1. INTRODUCTION

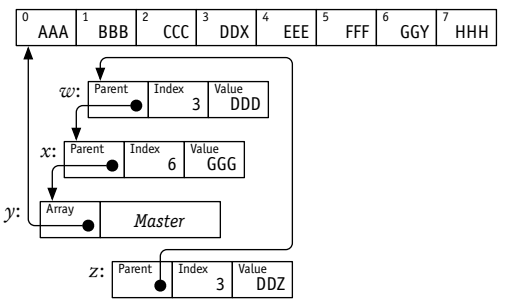
A classic method for efficiently storing multiple versions of a large file or data structure is to store just one version in full, and encode all other versions as *deltas* that express the changes between one version and the next. Such an approach is frequently used for external storage; for example, revision control systems have used this approach for more than twenty years [15]. The same techniques can and have been applied to arbitrary in-memory data structures [2, 13, 6]. Arrays are especially amenable to this form of storage compression because array update changes a single element of a potentially large array. Thus, if we wish to keep multiple versions of an array, we simply need to record which element updates need to be performed to transform one array version into another.

Strangely, work in this area (i.e., *persistent data structures*, defined in Section 2) almost always assumes that it is necessary to keep all versions of the data indefinitely—sidestepping the question of whether such data structures are actually amenable to garbage collection. Similarly, the reachability heuristic used by most garbage collectors does not work well on such data structures—if some of the versions present in one of these delta-based data structures are no longer required, the garbage collector is unlikely to realize that they are garbage.

SPACE 2006 Charleston, South Carolina, USA



1: An example trailer array.

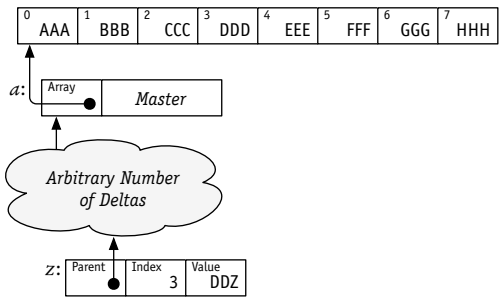


2: The example from Figure 1 after making y the root.

In this paper, we examine the case of *trailer arrays*. A trailer array, as described by Aasa, Holmström & Nilsson [1], is a collection of deltas and one actual array. Each delta represents a particular array version, usually created by an update performed on some previous version. Deltas store the array index that was modified, the new value for that array element, and a pointer to its "parent", which contains the values for all other array elements. The collection of deltas forms a tree in which pointers run from child to parent.[1] The delta at the root of the tree, the *master array*, is special, holding a pointer to an actual array instead of an array index and value. A simple trailer-array tree representing four versions of an array is shown in Figure 1.

While changes can be made in $O(1)$ time, reading an element may require traversing an arbitrary number of deltas.

[1]Normally, in a directed tree, there are pointers from parent to child, but throughout this paper we use terminology based on the underlying undirected tree.



3: Arbitrary garbage can exist between two needed array versions.

For this reason, the array can be *rerooted*, as shown in Figure 2. In this process, some delta is chosen to be the master array and each object between the old and new roots is changed to maintain the versions correctly. Rerooting is usually performed each time an array element is read.

Although a traditional tracing garbage collector may be able to remove some garbage from a trailer array, there can easily exist deltas that are logically garbage while nevertheless being "reachable" from the collector's perspective. In Figure 2, for example, z is easy to collect if it is unreferenced; but if z remains referenced and w is unreferenced by the program, the pointer from to z to w would cause w to be retained even though it is garbage. Moreover, whether w is garbage depends entirely on its value, not on which nodes it points to—if w modified element 6 rather than element 3, it would not be garbage.

Retaining garbage is a potentially serious issue. If the values stored in the array could be large (e.g., strings), retaining even one delta may be problematic. In the worst case, an arbitrary number of deltas may be retained, as shown in Figure 3. This latter scenario can occur if we have an array that we are making changes to and at some point we decide to hold on to an old version. We may then continue making changes to the array and build up a long chain of deltas between the two versions, of which at most n could actually be necessary for an array of size n. Moreover, there is no guarantee that this garbage will be collected in future garbage-collection cycles.

Our contributions in this paper are as follows:

- We raise the issue of collecting change-based structures, which seems to be rarely, if ever discussed.
- We present a method for garbage collecting trailer arrays. In the absence of cyclic dependencies, our method has complexity linear in the size of the data structure—a complexity equivalent to standard tracing collection. Unlike standard tracing collectors, our algorithm can recognize reachable parts of the trailer array as garbage and free them.
- We present a proof that our method is correct.
- We present an efficient set representation that may be useful for related problems, and show that it requires constant space and has amortized constant-time performance.

2. BACKGROUND

Data structures in languages such as C, Java, and Python are usually *ephemeral* [6], meaning that changes to the structure are destructive—the old version is lost and only the new version incorporating the change remains. In contrast, making a change to a *persistent* structure creates a new way to access the structure that displays that change, but the structure will appear unchanged from any previous reference to it.

Persistence is a useful property. It is a requirement for data in a purely functional language, because such languages disallow mutation of existing data. Persistent data structures are also used in the imperative world for storing moving Zand-et-al:92:mvng-images:sac [16], editing digital Nugroho-Sajeev:95:prsst-music:sac [10], and solving computational geometry problems [14, 7].

Although functional data structures are, by definition, persistent, the converse is not true—persistent data structures need not be implemented using functional-programming techniques. A persistent data structure only needs to ensure that what can be externally observed does not change—it can use mutation to adjust itself behind the scenes. For example, the rerooting performed in Figure 2 does not change the visible contents of the array versions over their contents in Figure 1, but internally the data structure has changed considerably.

Persistent arrays impose the requirements of persistence on arrays, transforming array update from a side effect that modifies an existing array into a function that appears to return a whole new array with a single element changed. Persistent arrays are of interest not only because persistence makes new array algorithms possible, but also because functional languages require or prefer that their data structures be persistent.

Unfortunately, there appears to be no perfect way to provide persistent arrays. *Trailer arrays* [2, 1, 3, 4] are a popular choice with a particular set of trade offs. The method is easy to implement, requires constant space for single-element updates, and supports many common access patterns in constant time or constant amortized time (see Section 7 for a comparison with other techniques).

3. CONCEPTUAL ALGORITHM

We will begin describing our method for garbage collecting trailer arrays with a conceptual overview of the algorithm. The algorithm has multiple phases of execution. The first three phases discover everything that must be kept. The fourth and final phase prepares the tree for the removal of unneeded deltas.

In our discussion, we will refer to the following data stored in a delta x:

- $\mathsf{index}(x)$ — The index of the array element changed by the delta.
- $\mathsf{value}(x)$ — The new value for that element.
- $\mathsf{parent}(x)$ — The parent of x; that is, the array version that this delta changes (if $\mathsf{parent}(x)$ = `NULL` then x is the root of the tree and stores the pointer to the ephemeral array).
- $\mathsf{indegree}(x)$ — A count of the number of pointers to x that have not yet been processed by Phase 3 (the

count begins at zero, rises to the in-degree of the node in Phase 2 and falls back to zero in Phase 3).

- ignored(x) — An optional temporary store for the set of array indices that are ignored by all x's children examined so far (initially no set is stored, represented by the undefined value).
- needed(x) — A single bit indicating whether this delta is needed, initially false (i.e., a mark bit).

In addition, the algorithm uses two sets, both initially empty:

- L — The set of externally reachable deltas (i.e., those the program explicitly needs because it needs that array version).
- T — The set of array indices that are ignored by all deltas in the current subtree.

Phase 1: In this phase, we find L, the set of externally reachable deltas. The work of this phase is mostly done by the garbage collector, using whatever method it uses to find live data. The only change to normal operating procedure is that each delta x is treated as atomic by the collector (the object is not examined for pointers) and we add each x to L. Garbage collection continues until the only unexamined objects are deltas.

Phase 2: In the second phase we find the in-degree of each of the reachable deltas and store it in their indegree counters. We do so by traversing the path from each node $x \in L$ to the root, starting at x. At each node y on the path, we increment indegree(y) and then,

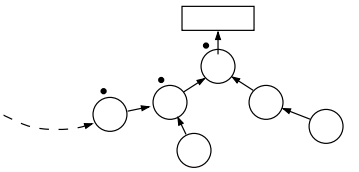
- If indegree(y) > 1, stop traversing the path. Continue with the next path.
- If indegree(y) $= 1$, move on up the path (i.e., $y := parent(y)$) and repeat.

This phase is illustrated in Figure 4; in this diagram the indegree counter of each node is shown by the number of dots placed outside it.

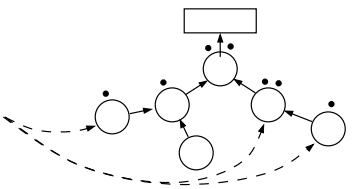
Phase 3: The goal of Phase 3 is to mark as needed every delta that contains a change required for some version of the array in L. To accomplish this task, we maintain sets of array indices that track which indices are unnecessary at a node. We keep one active set, T, and at times we store T in the ignored field of a delta.

For each node, $x \in L$, we traverse the path from x to the root. We begin the path traversal with $T := \varnothing$. For each node y on this path, starting with x, if ignored(y) is defined, we perform the following updates: $T := T \cap$ ignored(y) and ignored(y) := undefined. We intersect the sets because an index is only unnecessary if it is unnecessary for all versions of the array that depend on x. We then decrement indegree(x), and

- If indegree(y) > 0, we stop traversing the path at that point (because we will reach y again later coming from a different x). We set ignored(y) := T and $T := \varnothing$. We then continue with the next path traversal.
- If indegree(y) $= 0$, we process y as follows:
  - If index(y) $\in T$, the delta y is not needed and so we do nothing.



(a) After one node from L has been processed



(b) After three nodes from L have been processed

4: An example trailer array during Phase 2. The modification indices and values are not shown, only the parent pointers.

  - If index(y) $\notin T$, y is necessary and we have determined that value(y) is live and needed(y) := true.

  To continue along the path, we update $T := T \cup \{\text{index}(y)\}$ and $y :=$ parent(y).

We follow a special procedure to process the root of the tree, r, because it is a special node that has no index field. Each element of array(r), with index i, is live if $i \notin T$—thus, if $T = \varnothing$, all of the elements in the master array are live. Finally, we set ignored(r) := T and $T := \varnothing$.

Phase 4: At the start of this phase, we have found all the live data, and we now begin the process of cleaning up the data structure to remove deltas that we do not need.

In this phase, we fix the ephemeral array at the root of the tree to ensure that all of its elements are required, potentially eliminating the need for some deltas and avoiding any wasted space in the array. If ignored(r) $= \varnothing$, there is nothing to do and we can proceed to the next phase.

Otherwise, let r be the root of the tree (trivially available from Phase 3), and arbitrarily pick an $x \in L$ (such as the last one from Phase 3) and traverse the path from r to x (i.e., the reverse of our usual order). For each y on the path, if index(y) $\in$ ignored(r), then set array(r)[index(y)] := value(y) and needed(y) := false.

Notice that moving values in ignored(r) off our chosen path into array(r) has no effect on other paths to r, because, by definition, they ignore whatever value the r contains at those indices. Also, it might seem like it would be awkward to trace the path backwards from r to x, either requiring a stack or something akin to rerooting, but, as we shall see in Section 4, this process is actually trivial, given ignored(r).

Phase 5: In this phase, we wish to remove all the garbage deltas (taking care to properly preserve the tree structure). A delta g is garbage if needed(g) = false, and is live otherwise. Preserving the tree structure requires that we adjust the parent pointer of each nongarbage delta to skip any garbage deltas and point to its first live ancestor.

First we repeat the work we performed in Phase 2 to properly set the indegree counters for all nodes, but this time we will merely use the zero/nonzero status of the indegree to

indicate done/to-do. To avoid confusion, we will now refer to the "indegree" field as the "fixed" field, with values done and to-do. After applying the Phase 2 algorithm, all deltas reachable from the nodes in L have a nonzero indegree field and thus are marked to-do.

We traverse the tree, following the path from each node in $x \in L$ toward the root, by applying the following recursive algorithm to x: To fix a node y, if $\mathsf{fixed}(y) = \mathsf{done}$, we have already fixed this node. Otherwise, we first recursively fix $\mathsf{parent}(y)$, and then, if $\mathsf{needed}(\mathsf{parent}(y)) = \mathsf{false}$, we set $\mathsf{parent}(y) := \mathsf{parent}(\mathsf{parent}(y))$. Finally, we set $\mathsf{fixed}(y) := \mathsf{done}$.

Although we have presented the above algorithm recursively, it is straightforward to implement it nonrecursively using pointer reversal.

3.1 Correctness

We now wish to prove the correctness of our algorithm. Most importantly, we wish to show that it only deletes garbage, but we will also show that it deletes *all* garbage contained in deltas.

Let us first define the sets of items that we will be considering in the proof. We call the set of all deltas X. L is as defined earlier (i.e., all deltas reachable from external data). For modification cells $x, y \in X$, we define $x \to y$ to mean that $\mathsf{parent}(x) = y$ (i.e., that x points to y) and $x \xrightarrow{*} y$ to mean that x is a descendant of y (i.e., there is a directed path from x to y). We include x among its ancestors, so $x \xrightarrow{*} y$ means that y is reachable from x within the structure. We next define a set R_v for each $v \in L$ where $R_v = \{x | v \xrightarrow{*} x\}$ and a set $R = \bigcup_{v \in L} R_v$. We then define the set of unreachable deltas as $U = X \backslash R$. The set I contains all array indices. Because the set T changes over time, we use T_x to refer to the value of T when the algorithm is about to process node x. This definition leads us to our first item:

LEMMA 1. $\forall x \in L : T_x = \varnothing$

PROOF. We begin at each x with an empty tracking set. If we do not immediately process x, we still intersect each set that comes in with the stored empty set resulting in a new empty set. □

The algorithm also creates a set $T_x \cup \{\mathsf{index}(x)\}$, and it is convenient to give that set a name, so we shall.

DEFINITION 2. $T_x^+ = T_x \cup \{\mathit{index}(x)\}$

LEMMA 3. $\forall x \in R, x \notin L : T_x = \bigcap_{c \in \{c \mid c \in R, c \to x\}} T_c^+$

PROOF. This lemma follows directly from the algorithm. For reachable internal nodes, the tracking set is formed by progressively taking the intersection of the T_c^+ sets of all of its (reachable) children. □

For completeness, we define T_x for $x \notin R$ (i.e., unreachable nodes we never visit). In spirit, this definition follows from the above lemma, because such a node has no reachable children, and it seems reasonable that $\bigcap_{c \in \varnothing} T_c^+ = \mathscr{U}$, the universal set. We define it as I, the set of all indices.

DEFINITION 4. *For each $x \in U$, $T_x = I$.*

We next define our garbage. We say, for some $v \in L$, $x \in X$, that v needs x if accessing $\mathsf{index}(x)$ from v uses the data in x.

DEFINITION 5. *v needs x iff $x \in R_v$ and $\forall y \in [v, x)$: $\mathit{index}(y) \neq \mathit{index}(x)$ (where $[a, b)$ is used to indicate the set of nodes that are on the path between a and b, including a but not including b).*

At a node we do visit, we mark the node as live based on whether its index is needed.

LEMMA 6. *x is marked as live iff $\mathit{index}(x) \notin T_x$*

PROOF. This lemma follows directly from the algorithm. □

Next, we examine the relationship between tracking sets in a node and that node's parent.

LEMMA 7. *If $i \notin T_x$ and $\mathit{index}(x) \neq i$ then $i \notin T_{\mathit{parent}(x)}$.*

PROOF. Again, this lemma follows from our algorithm. From Definition 2, if $i \notin T_x$ and $\mathsf{index}(x) \neq i$, then $x \notin T_x^+$. Because $T_{\mathsf{parent}(x)}$ is formed either according to Lemma 1 or Lemma 3, $i \notin \mathsf{parent}(x)$. □

LEMMA 8. *If $i \notin T_x$ then $x \in L$ or $\exists c \in \{c \mid c \in R, c \to x\}$ such that $i \notin T_c$ and $i \neq \mathit{index}(c)$.*

PROOF. If $x \in L$, then $T_x = \varnothing$, and thus $\forall i \in I, i \notin T_x$. If $x \notin L$, then from Lemma 3, $T_x = \bigcap_{c \in \{c \mid x \in R, c \to x\}} T_c^+$ and thus $\exists c \in \{c \mid c \in R, c \to x\}$ such that $i \notin T_y^+$. □

We now have the pieces necessary to prove the correctness of our algorithm. We will begin by proving that our algorithm only deletes garbage, and then show that it deletes all garbage deltas.

THEOREM 9. *For all $x \in X$, if $\exists v \in L$ such that v needs a, then a is not deleted.*

PROOF. Assume there exists such a v. By Lemma 1, $T_v = \varnothing$ and thus $\mathsf{index}(x) \notin T_v$. By Definition 5, $x \in R_v$ and $\forall y \in [v, x)$: $\mathsf{index}(y) \neq \mathsf{index}(x)$. By induction via Lemma 7, $\mathsf{index}(x) \notin T_x$. By Lemma 6, x is marked. Therefore x will not be deleted. □

THEOREM 10. *For all $x \in X$, if $\nexists v \in L$ such that v needs x, then x is deleted.*

PROOF. We will prove this theorem by contradiction. Assume that x is not deleted. Then x was marked, and so, by Lemma 6, $\mathsf{index}(x) \notin T_x$. Thus, by Lemma 8, either $x \in L$ or $\exists c \in \{c \mid c \in R, c \to x\}$ such that $i \notin T_c$ and $i \neq \mathsf{index}(c)$. We can repeat this process on $\mathsf{index}(x) \notin T_y$, but because the structure is finite, we must eventually find some $z \in L$. Because $\mathsf{index}(x)$ is not equal to $\mathsf{index}(y)$, $\mathsf{index}(z)$, or the index of any node in between, and because $x \in R_v$, we can apply Definition 5. Thus z needs x, and therefore $\exists v \in L$ such that v needs x. □

4. IMPLEMENTATION

We now will present details of our implementation of the algorithm described in Section 3 for garbage collecting trailer arrays. These details are important in achieving the desired linear complexity.

4.1 Data Structures

In the algorithm, we make use of various data structures to maintain the needed information, including a set list of reachable nodes, L, and sets of array indices. We have created special implementations of these structures both because we wish to avoid allocating memory during garbage collection, and because we desire a linear-time algorithm.

4.1.1 The Externally Reachable Deltas

In every phase of the algorithm we make use of a set, L, of deltas. Because we only use this set to iterate over its contents in arbitrary order, we can represent it as a linked list.

We preallocate an extra pointer $\mathsf{liveset}(x)$ in each delta x and keep two static pointers, $head_L$ and $tail_L$. We can then insert delta x into the list at the head by setting $\mathsf{liveset}(x) = head_L$; followed by $head_L = x$ (and setting $tail_L = x$ if x is the first node to be inserted). We can also determine whether x is already in the list (since we don't want to insert it twice) by examining $\mathsf{liveset}(x)$ to see if it's `NULL`; and, if so, comparing x to $tail_L$ because the only item in L with a non-`NULL` liveset pointer is the last element.

4.1.2 Sets of Indices

Phase 3 depends heavily on being able to maintain sets of array indices. We need to maintain both a "current set" T and also store sets at deltas. We will implement these sets using more linked lists of deltas. Each delta x will have a bit $\mathsf{hasset}(x)$ to indicate whether or not a set is stored at that object. x will also have a pointer $\mathsf{elem}(x)$ to point to either the first element in the set stored at x (when $\mathsf{hasset}(x) = \mathsf{true}$) or the next element in the set (when $\mathsf{hasset}(x) = \mathsf{false}$). Recall that when we store a set at x we do not use $\mathsf{index}(x)$ in the set. Each x will, by default, have no set stored so we initialize $\mathsf{hasset}(x) := \mathsf{false}$ and $\mathsf{elem}(x) :=$ `NULL`.

To provide the necessary complexity, the set T is represented by two separate representations that are kept in sync. The values in T are represented by a linked list, whose head is stored in $head_T$, and also by a bit vector B with size equal to the size of the ephemeral array. We allocate a bit vector B as long as the ephemeral array when we create a trailer array. This bit vector can be stored with the ephemeral array and found during Phase 1 when the array is traversed.[2] The set T is initially empty, so we initialize the elements of the bit vector to false when it is created.

Let us consider the operations on sets of indices performed by our algorithm:

- $i \in T$

  Simply check $B[i]$. $O(1)$ time is required.

- $T := T \cup \{\mathsf{index}(y)\}$

  Set $B[\mathsf{index}(y)] := \mathsf{true}$, $\mathsf{elem}(y) := head_T$, $head_T := y$. *Assumes that y is not being used to store any set, which is true in our algorithm.* $O(1)$ time is required.

- $T := \varnothing$

  We traverse the linked list, starting at $head_T$. For each n in the list, set $B[\mathsf{index}(n)] := \mathsf{false}$ and $\mathsf{elem}(n) :=$ `NULL`. $O(|T|)$ real time is required.

- $\mathsf{ignored}(y) := T$ and $T := \varnothing$

  We store T by iterating over T, setting the corresponding bits in B back to false. Then set $\mathsf{elem}(y) := head_T$ and $\mathsf{hasset}(y) := \mathsf{true}$, and $head_T :=$ `NULL`. $O(|T|)$ real time is required.

- $T := T \cap \mathsf{ignored}(y)$ and $\mathsf{ignored}(y) := \mathsf{undefined}$

  First, note that the bits set in B correspond to each element of T. We now iterate over $\mathsf{ignored}(y)$, setting $B[\mathsf{index}(n)] := \mathsf{false}$ for each $n \in \mathsf{ignored}(y)$—the first such n is found in $\mathsf{elem}(y)$ and the next node after n is found in $\mathsf{elem}(n)$. As we iterate, we set $\mathsf{elem}(n) :=$ `NULL`, and afterwards set $\mathsf{hasset}(y) := \mathsf{false}$. At this point, the only bits that will be true in B are those that are *not* in $\mathsf{ignored}(y)$. We then create the intersection by iterating over the linked-list representation of T once more, flipping the bit $B[\mathsf{ignored}(m)]$ for each m in T and removing m from the list if its corresponding bit ends up as false. $O(|T| + |\mathsf{ignored}(y)|)$ real time is required.

[2]Alternatively, we could allocate a static bit vector large enough for the largest trailer array.

4.1.3 Additional data

The items listed above account for most, but not all, of the data for which we must preallocate space. Each delta also needs the indegree counter preallocated.

5. COMPLEXITY

Scanning and compacting a trailer array using our method has complexity comparable to normal garbage-collection algorithms; that is, linear in the size of memory being examined. To prove this result, we must show that the number of node visits performed by each phase of algorithm is linear in the total number of reachable nodes and that a constant amount of work is done at each node. Most important is showing that the set operations take no more than constant amortized time.

For most trailer arrays, a single scan of the array is sufficient; but if trailer arrays are allowed to hold arbitrary data, cyclic dependencies become possible. If array versions directly or indirectly refer to older array versions that were not otherwise reachable, multiple scans may be required. We will ignore this issue for now, and examine it in more detail in Section 6.

5.1 Phases

In each phase, we follow each pointer into a reachable node once, so our sweep visits a number of nodes bounded by the number of edges into the nodes.[3] These edges can either be from other nodes or from elsewhere in memory, in which case we find them in L. Since each node has out-degree 1, the number of edges from other nodes is bounded by the number of nodes. Similarly, each node can only appear once in L and thus the number of edges from elsewhere in memory for which we do work is also bounded by the number of nodes.

5.2 Sets

We now demonstrate that all required operations on these sets can be done in constant amortized time. We will be using the accounting method for amortized analysis. We will maintain the following invariants:

- Every set of size n will have n rupees in its account.[4]
- The set T will have an additional n rupees for a total of $2n$ rupees in its account.

[3]Since we cannot know about nonreachable nodes, we'll assume from now on that nodes we mention are reachable.

[4]Rupees are the currency of our local Amortized Analysis Bank.

Our sets have only a few restricted operations, as described earlier:

- $i \in T$

  This operation has constant real time, and no financial changes occur.

- $T := T \cup \{\mathsf{index}(y)\}$

  This operation has constant real time, but also requires a deposit of 2 rupees. Again, constant real and amortized time.

- $T := \varnothing$

  This operation requires $O(|T|)$ real time, but we have $2|T|$ rupees, so the operation is easily paid for, resulting in constant amortized time.

- $\mathsf{ignored}(y) := T$ and $T := \varnothing$

  This operation requires $O(|T|)$ real time, but we have $2|T|$ rupees. We spend $|T|$ rupees to pay for the operation, and save the other $|T|$ rupees for $\mathsf{ignored}(y)$. Again we have paid for the operation, resulting in constant amortized time.

- $T := T \cap \mathsf{ignored}(y)$ and $\mathsf{ignored}(y) := \mathsf{undefined}$

  Let us define $m = |T|$ and $n = |\mathsf{ignored}(y)|$. We begin with a total of $2n+m$ rupees from both sets. The operation requires $O(m+n)$ real time, which is equivalent to taking $O(\max(m,n))$ time. The new set has size at most $s = \min(m,n)$, due to the properties of set intersection. Thus we can use $\max(m,n)$ rupees to pay for this operation and we are left with $\min(m,n)+n \geq 2s$ rupees to put in our tracking set's account. We put in enough to maintain the invariant and potentially some to spare.

Thus, all operations require constant amortized time.

6. CYCLIC DEPENDENCIES

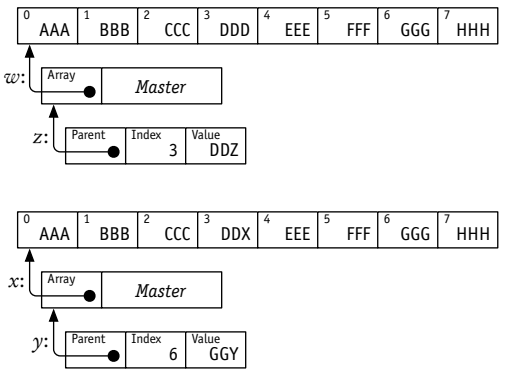
Throughout this paper, we have assumed that the structure was free of cyclic dependencies. But if array versions directly or indirectly refer to other array versions that were not otherwise reachable, multiple scans may be required. Phases 1–3 of our algorithm scan the array to determine what data is live and it is these phases that might need to be repeated. Phases 4 and 5 compact the trailer array and will always be executed exactly once.

Although it is possible to simply discard the work of previous scans and rescan the trailer array from scratch, it is straightforward to modify the algorithm to preserve prior work by making two observations:

1. Only deltas that were added to L since the last scan need to be examined in phases Phases 1–3.

2. We can stop scanning whenever we reach a delta that has already been processed *if* T was empty at that node (because from that node onwards, all indices were required).

Unfortunately, a suitably unpleasantly constructed array of size n could require $O(n)$ scans, each of which required $O(n)$ time to perform, resulting in $O(n^2)$ time performance for garbage collection. The only consolation is that such a strange array would be equally unpleasant to construct using actual arrays, and so the data structure we are representing would itself be grotesque.



5: The tree from Figure 1 after splitting between w and x.

But if a constant number of scans is performed, garbage collection remains linear in the size of the trailer array examined. One possible way to avoid poor time performance is to limit the number of scans on a particular trailer array to a small constant, and after that fall back to the mechanisms of traditional tracing collection and thus potentially pay a space penalty rather than a speed penalty.

7. RELATED WORK

Trailer arrays are just one technique for providing persistent arrays (sometimes called functional arrays). There are several other techniques that provide similar functionality using different (usually tree-based) data structures, including Okasaki's purely functional random-access lists [11]; Dietz's, and later O'Neill & Burton's refinement of, the fat-elements method [5, 12]; and Hinze's one-sided flexible arrays [8]. Each of these techniques makes different trade-offs for complexity, typically being logarithmic for array access and update, as compared to the constant-time performance that trailer arrays provide when updates are always performed on the most recent version—only O'Neill & Burton's fat-elements method also offers constant-time performance for such updates. Not all persistent-array data structures are prone to issues with garbage collection—those that are purely functional tend to avoid such issues.

Chuang [3, 4] suggests two possible enhancements to the basic ideas of trailer arrays, each of which can reduce the length of a chain of deltas at the cost of creating additional ephemeral arrays. Chuang observed that when the number of updates v is much greater than the size of the array n, even rerooting the tree does little to speed up array accesses. He speeds up accesses by breaking long chains of deltas, creating two distinct trailer arrays (see Figure 5), but this approach requires additional space for new copies of the ephemeral array. In his first paper [3], he waits until a long ($> 2n$) reroot is performed and then splits the tree after every $\Theta(n)$ deltas. This method has $O(n)$ amortized access time and increases memory usage only by a constant factor. In his second paper [4], Chuang uses a randomized approach in which every step of rerooting has a $1/n$ probability of splitting the tree. For a relatively large number of accesses this method will tend to make a copy of the ephemeral array at each heavily accessed delta, which will eventually provide

constant-time access but use $O(v \cdot n)$ memory.

The extent to which Chuang's trailer-array variations create garbage differs from basic trailer arrays, but each can still produce arbitrary amounts of uncollected garbage.

Many complex data structures have been designed to implement persistent versions of other useful ephemeral structures [13, 6, 9]. Like trailer arrays, these structures contain the information for all versions of the ephemeral structure they represent along with the necessary bookkeeping to find the data for the specific version being accessed.

8. CONCLUSION

Our algorithm collects all unnecessary garbage from trailer arrays in linear time (assuming no cyclic dependencies) with a constant-factor overhead for bookkeeping. This result is pleasing because we are able to collect this specialized data structure with the same time and space performance that is expected of ordinary garbage-collection techniques and we collect all garbage. We are able to collect all garbage because we can efficiently detect and remove all unnecessary/unreachable modification nodes from the data structure. Removing these modification nodes not only frees memory, but also increases the efficiency of access and reroot operations by shortening paths to the master array.

In addition to providing a method for garbage collecting trailer arrays, our algorithm can be used to manually remove undesired versions. Removing versions from persistent data structures is underdiscussed in literature, and never discussed specifically for trailer arrays. We feel that having this extra operation makes the trailer-array data structure complete. Now we no longer need to adhere to the unspoken rule that persistent data structures must keep track of all versions; we can still reap their benefits by keeping only those versions we need.

Finally, we have shown how trailer arrays provide an example of a simple data structure where traditional tracing collection does not discover all logical garbage. Instead, a custom algorithm that understands the semantics of the data structure is required—the topological structure of the data is not sufficient to accurately determine liveness.

9. FUTURE WORK

There is still much work to be done in the area of garbage collecting persistent data structures. Trailer arrays are not the only peristent data structure that keeps track of changes in separate nodes. Overmars [13] showed that any data structure can be made persistent by keeping track of the changes in separate nodes. One direction for future research is to develop a mechanism that is general enough to correctly garbage collect any persistent data structure that stores changes according to Overmars's method, not just trailer arrays. We are also interested in garbage collecting the persistent linked data structures of Driscoll et al. [6]. Driscoll chooses to store changes alongside original data within preexisting nodes, differentiating between updates by giving each data field a label called a "version stamp". Driscoll's method for maintaining persistence is very different from Overmars's, requiring entirely new algorithms to be developed. Finally, at this time the most feasible method for implementing these algorithms is to build a plug-in (i.e., a hack) for an existing garbage collector. Having a "hack" for every special data structure is undesirable, especially when it needs to be applied to every garbage collector. The most important direction for our research is in developing an extension to ordinary garbage-collection techniques that allows for the collection of persistent data structures, as well as other structures that are hard to garbage collect.

10. ACKNOWLEDGMENTS

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

11. ADDITIONAL AUTHORS

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

12. 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] H. G. Baker. Shallow binding in LISP 1.5. *Communications of the ACM*, 21(7):591–603, July 1978.

[3] T.-R. 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.

[4] 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.

[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] M. T. Goodrich, M. Orletsky, and K. Ramaiyer. Methods for achieving fast query times in point location data structures. In *SODA '97: Proceedings of the 8th Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 757–766, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[8] R. Hinze. Bootstrapping one-sided flexible arrays. In *ICFP '02: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 2–13, New York, NY, USA, 2002. ACM Press.

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

[10] L. E. Nugroho and A. S. M. Sajeev. Persistence in music data structures. In *SAC '95: Proceedings of the 1995 ACM symposium on Applied Computing*, pages 27–31, New York, NY, USA, 1995. ACM Press.

[11] C. Okasaki. Purely functional random-access lists. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 86–95, La Jolla, California, 25–28 June 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

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

[13] M. H. Overmars. Searching in the past II: General transforms. Technical Report RUU–CS–81–9, Department of Computer Science, University of Utrecht, 1981.

[14] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, July 1986.

[15] W. F. Tichy. Design, implementation and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering*, Tokyo, Sept. 1982. IEEE.

[16] M. K. Zand, H. Farhat, and H. Saiedian. Persistent structures to store moving images. In *SAC '92: Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing*, pages 21–27, New York, NY, USA, 1992. ACM Press.