Garbage Collecting the Split Node Persistent Data Structure

Jonathan Beall David Malec Phil Miller Ari Wilson
Melissa O'Neill

September 20, 2007

1 Introduction

Traditional garbage collection doesn't work, because some data structures are hard to garbage collect effectively [Boehm, 2003, Effinger-Dean et al., 2006a,b]. In this paper we examine the structures created by the node splitting persistence algorithm of Driscoll et al. [1989]. *Persistent data structures* are those in which updates create new versions and accesses or updates to the structure can apply to any version in the tree of all versions. Persistent data structures are useful for certain computational geometry algorithms [e.g., Sarnak and Tarjan, 1986, Gupta et al., 1994], undo operations in structured editing [Reps et al., 1983], and in implementing data structures for functional programming [e.g., Erwig, 1998, Okasaki, 1998].

There are many methods for implementing persistent data structures. The naïve approach duplicates the entire structure in every version. Purely functional programs share some common substructure while duplicating parts that vary. More efficient methods store differences between versions and avoid duplicating data common to multiple versions [e.g., Overmars, 1981a,b, Driscoll et al., 1991]. Thus, any version can be generated as needed while avoiding the storage costs of the naïve approach. Driscoll et al. have proposed the node-splitting method to make any linked data structure of constant-bounded in-degree persistent. This method is currently the most efficient for a broad class of data structures. We describe split node structures in Section 2.

In the literature of persistent data structures, there is a common implicit assumption that all versions should be remembered permanently. However, this assumption is not well-founded in programming practice. Programmers justifiably expect that memory allocated to store intermediate results will be reclaimed when those results are no longer useful. Similarly, users of persistent data structures should not have to pay the memory cost of intermediate versions of a data structure for the life of that data structure.

The naïve approach to persistence makes garbage collection of forgotten intermediate versions and inaccessible nodes trivial, but at the

cost of high space and time overheads on updates to the persistent structure. Unfortunately, the more efficient node-splitting method creates an extensive network of pointers to achieve its performance. Traditional collectors (which use the conservative pointer-reachability heuristic for data liveness [as described in Wilson, 1994]) will never garbage collect any of this information.

We improve on this situation in the following ways:

- We describe an algorithm to identify nodes and pointers in a split node structure that are in dead versions or obviously inaccessible in Section 4.2 in linear time and constant space.
- We give several algorithms to identify nodes and pointers in a split node data structure that are either in dead versions or inaccessible in Sections 4.1, 4.3, and 4.4. The algorithms differ in their run time and completeness guarantees.
- We give a cleanup algorithm that sweeps all identified garbage in Section 5. We specify the algorithm and prove that it maintains all invariants which were preserved by the node-splitting data structure in Section 5.
- We give evidence that our algorithms can be implemented efficiently. We do so in the form of node-splitting persistent deques and binary search trees in Python in Section 6.
- We conclude by explaining some open problems in garbage collecting the Driscoll-Tarjan data structure in Section 7.

2 The Node-Splitting Method

We wish to garbage collect the Driscoll et al. split node structure. However, some background on the data structure is necessary to understand our algorithms for garbage collecting it. We do not show the algorithms for creating and maintaining the data structure because they do not directly influence our methods or results. As a motivating example, we construct a persistent stack out of a simple non-persistent linked-list based stack.

2.1 The Ephemeral Structure

In our non-persistent stack s, pictured in Figure 1, each element contains a value, and a `next` pointer leading down into the stack. Any non-persistent data structure like s is termed an *ephemeral data structure*. Each element of s is referred to as a *node* and is composed of values and pointers fields. The pointer `top` is the only external entry point into s, and is referred to as s's *access pointer*.

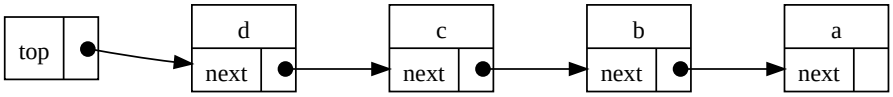

Figure 1: Ephemeral stack.

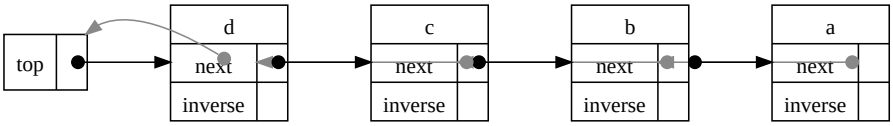

Figure 2: Ephemeral stack augmented with inverse pointers.

In our example, the only operations performed were the pushes of 'a', 'b', 'c', and 'd', which created four new nodes in s. In general, accessing or updating information in an ephemeral data structure is done by starting at an access pointer and then following pointers, reading data, changing data, and creating new nodes as needed. Note that, because the number of access pointers is fixed, we can treat the structure as having a single access pointer that acts as a node containing the real access pointers.

From a programmer's perspective, a persistent data structure rendition of an ephemeral data structure and a list composed of individual copies of each prior version of the same ephemeral data structure should behave identically. This property is known as the *persistence abstraction*. Maintaining this abstraction during updates to the structure requires some bookeeping work. This work sometimes includes modifying all pointers to a node. The method thus requires that each ephemeral node be augmented with inverse pointers to record which nodes point to it. Since the node-splitting method is applied to a linked ephemeral data structure with bounded in-degree for each node, we can bound the number of inverse pointers we will ever need for a given node. The node-splitting method thus requires that each node have as many inverse pointers as the maximum indegree for a node, including access pointers. For our ephemeral stack, each node is pointed to by either the `top` access pointer or another node's `next` pointer. We have a maximum indegree of one and so add one inverse pointer per node, as shown in Figure 2.

Whenever a pointer in the data structure is changed we update the relevant inverse pointers. Likewise, when an access pointer is changed, we update the relevant inverse pointers to clear or add labels referring to the access pointer. The rest of the node-splitting method treats inverse pointers and other pointers identically.

2.2 The Persistent Structure

Making an ephemeral data structure persistent requires that some additional storage be allocated to represent changes to the data structure. In the split node method, this storage is present in the form of linked lists of nodes, called *families*, that each logically represent one ephemeral node and associated changes. To maintain the persistence abstraction, several fields are added to each node in a family. These include:

- *Version stamp*: Indicates the first version for which a node is vald.
- *Copy pointer*: Links to the next node in a family after a node.
- *Extra pointers*: Stores some constant number of updates to pointers in a node without creating new nodes.

Each pointer also has a version stamp.

All nodes and pointers are associated with some interval of the version list within which they are valid, referred to as their *valid interval*. The interval for a node or pointer starts with its version stamp and continues until it is supplanted by a copy with a later version stamp. If no such later version exists, then it continues until the end of the version list.

Changes to the data structure are saved in one of two ways. Updates to pointers may be stored in the extra pointers contained within a node, if there are enough empty extra pointers. If there is not enough room, a copy of the node will be created with the new pointer saved inside. Value updates are always stored in copies of the node. These copies are linked together in version order within a family.

The ability to determine which versions descend from a given version is the final consideration in making a data structure persistent. This ability allows updates to particular versions to be easily located at a later time. Driscoll et al. use an ordered list data structure proposed by Dietz and Sleator [1987] to keep track of the versions of the data structure. The ordered list data structure allows us to add or subtract versions from the version list and to determine the relative order of two versions, both in $O(1)$ time. The version list is constructed to be a preorder traversal of the version tree. Therefore, the descendants of a version occur consecutively immediately after it in the version list.

Consider Figure 3. The nodes marked 'top' represent the root set of this persistent stack and comprise its access pointers. The values 'a' through 'd' were pushed into the stack to create versions 1.0 to 4.0. Version 5.0 was created when the value 'e' was pushed. Because older versions of the data structure can be referred to and operated on, the program was able to pop from version 4.0 of the stack to create version 4.5 between versions 4.0 and 5.0. In order to maintain consistency, the node containing 'd' was duplicated into version 4.5, even though it has logically been removed from the data structure.

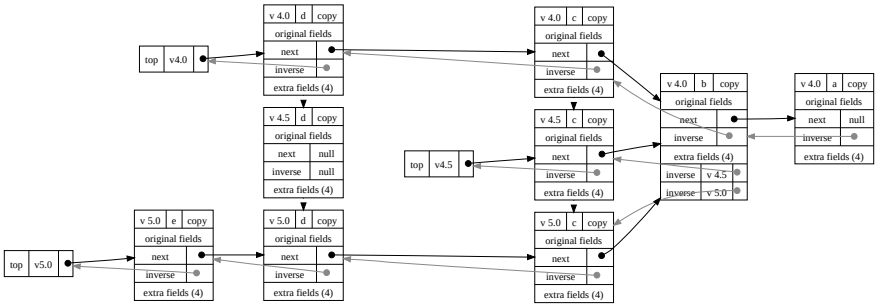

Figure 3: Our persistent stack, and the result of creating two new versions.

2.3 Invariants

To maintain the outside abstraction of persistence, the split node data structure preserves several invariants regarding pointers between nodes. These invariants are always preserved except in the course of an update operation. The data structure restores the invariants by adding values to extra pointers and by splitting nodes. Specifically, the data structure requires that pointers be:

Symmetric The symmetry property ensures that inverse pointers are properly maintained. The definition is slightly different for ordinary and access pointers.

- If a node x contains a pointer to a node y such that the valid interval of the pointer includes a version i, then y contains a pointer to x such that the valid interval of this pointer also includes i.
- The access pointer named a contains a pointer to a node x in position i if and only if node x contains a pointer to a with version stamp i.

Proper A pointer with version stamp i is proper if the pointer indicates a node x whose valid interval contains i.

Nonoverlapping A pointer with version stamp i is nonoverlapping if it indicates a node x, and its valid interval is contained within that of x. It is considered overlapping otherwise.

3 Algorithmic Overview

As mentioned in Section 1, a pointer-tracing collector will only reclaim nodes in the split node structure that have been completely unlinked. This unfortunate property can leave behind arbitrary amounts

of garbage in the form of no-longer-referenced versions and nodes and pointers that are inaccessible in the version(s) in their valid interval.

If it can be determined that some versions are no longer of interest, then it should be possible for the memory they occupy to be reclaimed. In Figure 3, consider the case in which versions 4.5 and 5.0 were created in a procedure call, which returned to the caller and allowed the only outside references to 4.5 and 5.0 to go out of scope. By pointer reachability, version 4.0 will cause the garbage collector to view as live all of the nodes that comprise versions 4.5 and 5.0.

As we mentioned in Section 2.2, node 'd' in version 4.5 is not reachable to the world outside the persistent stack. In order to see this, a tracing garbage collector would need to be familiar with the details of split node data structures. In lieu of this specialization of the garbage collector to specific data structures, we propose a series of algorithms that run within a framework in which the garbage collector traces to entry-points of the structure of interest, hands off the list to a collection procedure specified in the structure's definition, and then continues its trace through the structure and to any objects referenced therein. As in that paper, our analysis assumes that there is nothing in the structure that the garbage collector should treat as an entry point. If such dependencies are allowed, then any specialized collection algorithm might need to run $O(n)$ times in order to locate all live data.

We also assume that the mutator is not updating the split node structure at the time we wish to run our algorithms over it, as our algorithms assume that the tree is in a consistent state. This problem can be overcome in practical implementations by preallocating the amount of space necessary for the update to the structure, allowing the garbage collector to run safely before any disruptive changes can be made, or with the use of a lock bit.

Broadly speaking, our algorithms for garbage collection of the split node structure are split into a pair of phases analogous to the 'mark' and 'sweep' phases of a conventional collector. The *tracing* phase identifies which nodes and pointers remain accessible, and the *cleanup* phase removes everything else. We describe these algorithms using the following terminology, in which S is an arbitrary split node structure with version list V, of which some subset V_l has been identified as *live* by the primary garbage collector.

- A *versioned path* in S with respect to version v is a sequence of nodes such that for any consecutive pair of nodes a, b in the path, a has a pointer whose target is b and whose valid interval includes v. A versioned path is *rooted* if it begins at the access pointer.

- A node n is *reachable in version* v if there exists a rooted versioned path with version v to the node. A pointer in n is reachable in v if its valid interval includes v.

- A node or pointer x is considered to be *live version reachable* if there exists a live version v such that x is reachable in version v.
- A tracing algorithm on S is *correct* if it identifies all of S's live version reachable nodes and pointers as live.
- A tracing algorithm on S is *complete* if it never identifies any live version unreachable nodes or pointers in S as live.

In the asymptotic performance analyses of our algorithms, S's total size including nodes and pointers is denoted by n. V and V_l's sizes are defined to be v and v_l, respectively. Note that $v_l \leq v$ and that $v \in O(n)$ since each new version requires at least one access pointer. Since a traditional pointer reachability-based garbage collector takes $\Theta(n)$ time and space to trace the split node structure, we ideally want an algorithm that runs in the same time and space bounds while collecting all logical garbage.

We present a variety of approaches to tracing whose suitability depends on the space or time requirements of a particular application for persistence. The cleanup process described in Section 5 is carried out by a simple linear-time procedure that removes all nodes and pointers not identified as live during the tracing phase. The interface between our tracing algorithms and our generic cleanup algorithm is the function $reachedMin[x]$. $reachedMin$ maps a node or pointer x to the smallest version v such that x is reachable in v. For any node or pointer x that is not identified as live, $reachedMin[x] = \infty$.

3.1 Determining Next Live Versions

In order to determine a proper new version stamp for a node or pointer, some of our collection algorithms require the ability to look up the first live version at or after some particular version. We will denote this as $nextLive[v]$. This data can be stored in the version list for a constant factor space overhead, or it can be recorded in an auxiliary map from versions to live versions. In either case, $nextLive$ must be regenerated every time garbage collection occurs within the structure, as maintaining this map is only possible when the set of live versions is known. There is a simple $\Theta(v)$ algorithm to pre-compute $nextLive$ for every version, seen in algorithm *NextLiveList*.

3.2 Node Normalization

As mentioned in section 2, the split node structure maintains a number of invariants on the pointers between nodes. In order to make some proofs easier, we define a few more properties:

Compact The pointers in a node are compact if every pair of pointers with the same name and adjacent valid intervals have different targets.

Algorithm *NextLiveList*$(V, live)$
Input: V is the version list, $live[v]$ denotes whether version v is live, $next[last[V]] = \infty$

```
1.  v ←first element in V
2.  w ←first element in V
3.  while v < ∞
4.      do while ¬live[v] and v < ∞
5.             do v ←next[v]
6.         while w ≤ v and w < ∞
7.             do nextLive[w] ←v
8.                w ←next[w]
9.         v ←w
```

Algorithm *Coalesce*

```
1.  sort extraFields[n] by their version stamp
2.  for name in fieldNames[n]
3.      do f ←origField[n, name]
4.         for g in extraFields[n]
5.             do if name[g] ≠ name then continue
6.                if target[f] = target[g]
7.                   then remove g from n
8.                   else  f ←g
```

Exactly symmetric If a node x contains a pointer to a node y with valid interval I, then y contains a pointer to x with valid interval I.

Intuitively, compactness represents the absence of pointers that make no semantic contribution to the data structure.

There is a simple algorithm, which we call *Coalesce*, to make a given node's pointers compact. If this algorithm were run on every node in a structure, then the structure would exhibit exact symmetry. This algorithm runs in constant time, and cannot increase the space or time cost of operations on the data structure.

4 Tracing Algorithms

We now describe several algorithms to trace through the data structure while identifying all possibly-live information. Some of the algorithms run in time proportional to the size of the data structure, but don't collect all garbage. Others which run more slowly can pick up everything that's demonstrably no longer useful. Several auxillary functions required for our algorithms' operations are listed in Table 1.

Function	Definition
$accessPointers[s, v]$	Access pointer to split node structure s in version v.
$versionList[s]$	s's complete list of versions.
$target[p]$	Node pointed at by p.
$forwardPointers[n]$	List of the non-inverse pointers in a node n.
$validMin[x]$	Beginning of x's valid interval.
$validMax[x]$	End of x's valid interval.

Table 1: Function summary.

Algorithm *NaïveTracer*$(s, live)$
Input: s is a split node structure, $live$ is a descending sorted list of live versions. $reachedMin[x]$ is initially ∞ for all x except live access pointers.

```
1.  for v in live
2.      do ap ←accessPointers[s, v]
3.         NaïveVisit(ap, v)
```

4.1 Naïve Algorithm

The simplest approach to a correct and complete garbage collector for the split node data structure is to perform a version-checking depth-first search from each access pointer to find live version reachable nodes and pointers. This trace is specified in Algorithm *NaïveTracer*.

A non access pointer p is version reachable iff its containing node n is live version reachable and p's valid interval overlaps with the reached interval of n. Since v passes through all of $live$ in a strictly descending fashion, *NaïveTracer* will examine n in all of its live versions and mark p as reachable. Therefore, if n is marked correctly, its pointers will be marked correctly as well. Thus, in the following analysis, we only consider correct markings on nodes.

Algorithm *NaïveVisit*(p, v)
Input: p is a pointer in a split node structure that is reachable in version v.

```
1.  n ←target[p]
2.  if n is not nil and reachedMin[n] > v
3.    then reachedMin[n] ←v
4.         update reachability information with v for every pointer in
           n
5.         for q in forwardPointers[n]
6.             do if validMin[q] ≤ v ≤ validMax[q]
7.                    then reachedMin[q] ←v
8.                         NaïveVisit(q, v)
```

Theorem 1. *The naïve algorithm is correct.*

Proof. Consider a live version reachable node n in a split node structure S. By definition, there exists a live version v and a versioned path p conditioned on v from the access pointer in version v to n. In *NaïveTracer*, ap will eventually correspond to this access pointer. ap will then be passed to *NaïveVisit* along with v.

Since v is in each pointer in p's valid interval and since v is strictly decreasing within *NaïveTracer*, each node in p will be visited once during this top-level call to *NaïveVisit*. Thus, $reachedMin[n] < \infty$ and n has been marked live. □

Theorem 2. *The naïve algorithm is complete.*

Proof. Consider a live version unreachable node n in a split node structure S. Assume *NaïveTracer* has marked n as live. Then there exists a non-infinite version v such that $reachedMin[n] = v$. This means *NaïveVisit* has visited n at least once and thus there exists at least one pointer p such that $target(p) = n$. If p were an access pointer, this would be a rooted versioned path in v to n, which is a contradiction.

Therefore, p is not an access pointer. This argument is easily extended to show that none of the pointers in the versioned path leading to n are access pointers. However, since *NaïveVisit* always begins at an access pointer ap, it can never have found n and marked it as live. This is a contradiction. Therefore, the naïve algorithm is complete. □

Theorem 3. *The naïve algorithm runs in $\Theta(nv_l)$ time using $\Theta(n)$ space.*

Proof. Setting $reachedMin$ is the only extra space consumed by the naïve algorithm. Since $reachedMin[x]$ is set for all nodes and pointers that are live, and it is possible that all nodes and pointers are live, this can take up to $\Theta(n)$ space.

NaïveTracer runs in $O(nv_l)$ time because we start at each of v_l access pointers and there are only maximally n possible nodes and pointers we can access in a trace from any particular access pointer.

It also runs in $\Omega(nv_l)$ time. Consider the case where m nodes are created in an initial version, where $m \in \Theta(n)$. In the subsequent v_l versions, no modifications to existing nodes are made and no node nodes are created. These new versions cause a small chain of copies of nodes near access pointers to be created, to accomodate new inverse pointers to the access pointers. However, if $v \in \Theta(1)$, each trace from an access pointer will still take $\Theta(n)$ time. Since there are v_l access pointers, this example runs in $\Theta(nv_l)$ time.

□

4.2 Dead Versions Algorithm

The Dead Versions algorithm avoids tracing pointers through the data structure by assuming that any node or pointer that appears to be valid

Algorithm *DV*(s)
Input: s is a split node structure
1. $reachedMin \leftarrow \text{fn}(x) \Rightarrow$
2. let $v = nextLive[validMin[x]]$ in
3. **if** $v \leq validMax[x]$ **then** v **else** ∞

in a live version is reachable in that version. Thus, it can run very quickly in comparison to other algorithms presented. However, this speed comes at a cost: unbounded quantities of inaccessible nodes and versions in live versions can accumulate and remain uncollected. The algorithm is described in Algorithm *DV*.

Theorem 4. *The Dead Versions algorithm is correct.*

Proof. Consider a live version reachable node n in a split node structure S. Observe that in $reachedMin[n]$, v will be the least live version in n's valid interval, by the definitions of $validMin$ and $NextLiveList$. Sinve v is in the valid interval of n, $v \leq validMax[n]$. Hence, $reachedMin[n] = v$, and n is therefore marked live. □

Theorem 5. *The Dead Versions algorithm runs in constant time with constant space overhead.*

Proof. *DV* defines a function for $reachedMin$ that can be evaluated in constant time given the existence of $nextLive$. Since this function is only evaluated in *Cleanup*, *DV* runs in constant time. The function only takes constant space to store, so *DV* takes only a constant amount of space. □

4.3 Incremental Algorithm

The incremental algorithm for garbage collection involves modifying the naïve approach slightly in order to achieve the optimal $\Theta(n)$ time bound per collection run while collecting all garbage present at the beginning of a a sequence of v_l garbage collector runs. We give an outline of this approach in Algorithm *IncrementalTracer*.

This algorithm has higher intercollection requirements than other tracing methods. For example, the garbage collector must maintain the state of the variables v and $oldLive$ between runs. The cleanup algorithm also needs access to the variable v, so that it modifies only certain nodes and pointers. Those nodes and pointers have version stamps u such that $nextLive[u]$ conditioned on $oldLive$ is greater than or equal to v, meaning that no subsequent trace can affect the reachability of this object. The version list needs similar protection so that premature deletion of dead versions does not occur. Computing $reachedMin$ of nodes and pointers created after the beginning of a sequence of runs returns a special value ($-\infty$) so that these objects are not deleted during this sequence of runs.

Algorithm *IncrementalTracer*($s, live, oldLive, v$)
Input: s is a split node structure, $live$ is the current live version list in descending order, $oldLive$ is the live list at the beginning of a sequence of collection runs, v is the version we are examining, initially **nil**.

1. **if** v is **nil**
2. **then** $oldLive \leftarrow live$
3. $v \leftarrow$first element in $oldLive$
4. **for** all nodes and pointers x in s
5. **do** $reachedMin[x] \leftarrow \infty$
6. ($*$ Special value for objects that we can clean up after this sequence of collections. $*$)
7. **else** $v \leftarrow$version after v in $oldLive$
8. $ap \leftarrow accessPointers[s, v]$
9. *NaïveVisit*(ap, v)

In the following analysis, as in the naïve method, we only consider correct markings on nodes.

Theorem 6. *The incremental algorithm is correct.*

Proof. Consider a live version reachable node n in a split node structure S. There is at least one versioned path p conditioned on live version u starting at ap ending at n. Consider the maximum u with such a path.

Assume n is mistakenly considered dead. v is only changed once per run of *IncrementalTracer*, clean up happens once per run and only affects a node n with version stamp w if $nextLive[w] \geq v$. In the run when n is removed, v cannot be u, for, if it were, n would have been marked alive in that run of *NaïveVisit*, by the same argument as in the correctness proof for the naïve algorithm. Since u is live and in n's valid interval, $u \geq nextLive[w]$. Therefore, $u > v$. In that case, since we traverse the version list in descending order, there was a previous run starting at the access pointer in version u. That run would have found n and marked it alive. This is a contradiction, thus the incremental algorithm is correct.

□

Theorem 7. *After a sequence of v_l collection runs, the incremental algorithm will have picked up all garbage present before the initial run in the sequence.*

Proof. The argument here is identical to the proof of correctness for the naïve algorithm. □

Theorem 8. *The incremental algorithm runs in $\Theta(n)$ time using $\Theta(n)$ space.*

Algorithm *FastFullTracer*$(s, live)$
Input: s is a split node structure, $live$ is the live version list in ascending order, and Q is a priority queue, initially empty. $reachedMin[x]$ is initially ∞ for all x except live access pointers.

```
1.  for v in live
2.      do insert v into Q
3.  while Q is not empty
4.     do v ←front of Q
5.        if v has not yet had its access pointer pushed onto its stack
6.          then push accessPointers[s, v] on v's stack
7.          else  p ←pop from v's stack
8.                FastFullVisit(s, live, p)
9.        if v's stack is empty
10.         then dequeue v from Q
```

Proof. *NaïveTracer* ran in $\Theta(nv_l)$ time. We only look at one version each time, so *IncrementalTracer* runs in $\Theta(n)$ time. We take the same amount of space as *NaïveTracer*, except that we do need to maintain v and $oldLive$ between runs. Other algorithms can actually free this information as they have do not required data to be preserved between runs. □

4.4 Fast Full Algorithm

This algorithm improves upon the naïve algorithm by collecting all logical garbage in $O(n \log v_l)$ time and $\Theta(n)$ space. It does so by making use of a priority queue of live versions implemented as a binomial heap. Each live version has an associated stack in which pointers are stored. Pointers are only examined and marked when their last live version has been reached and all possible paths to them have been found. The full specification is given in Algorithm *FastFullTracer*.

In Section 4.1, we discussed why split node tracing algorithm analysis only requires examining correct markings on nodes. Since we store and examine pointers instead of nodes in this algorithm, we would here like to be able to analyze pointer markings rather than node markings. By the definition of versioned path, a node n is live reachable iff there exists a pointer p such that p's target is n and p is live. Since each pointer is examined at least once in *FastFullTracer*, each node will be marked correctly as well. In the following analysis, we thus only consider correct pointer markings.

Theorem 9. *The fast full algorithm runs in $O(n \log v_l)$ time using $\Theta(n)$ space.*

Proof. Each pointer in s can only be inserted once and this can cause at most one insertion and deletion from Q. Since Q's size is maximally v_l and operations on a binomial queue can only take up to $O(logn)$ time,

Algorithm *FastFullVisit*$(s, live, p)$
Input: s is a split node structure, $live$ is the live version list in ascending order, p is the pointer currently under examination. Q is a priority queue.

```
1.  n ←target[p]
2.  if n is not nil and reachedMin[p] < ∞
3.    then if reachedMin[n] = ∞
4.           then for q in forwardPointers[n]
5.                  do nl ←nextLive[validMax[q]]
6.                     push q on nl's stack
7.                     if nl is not enqueued in Q
8.                       then enqueue nl in Q
9.         reachedMin[n] ←min(reachedMin[n], reachedMin[p])
10.        update reachability information with p for every pointer in
           n
```

the overall run time for the algorithm is $O(n \log v_l)$. The proof of $\Theta(n)$ space usage is similar to the proof of the naïve method. □

5 Cleanup Algorithm

Once a trace has been performed, and $reachedMin$ is defined, we can traverse the data structure to unlink everything that the trace concluded was dead. This traversal eliminates a number of different kinds of garbage:

- Dead versions and their associated access pointers,
- Nodes and extra fields that are not live version reachable, and
- Extra fields that can be 'collapsed' into their containing node's original fields.

During the traversal, the version stamp on each live node or field is increased to the first live version in which that node or field is reached, thereby eliminating all references to dead versions. Afterwards, the resulting structure still meets the pointer invariants, and hence the asymptotic time and space bounds, that make the node-splitting method so attractive. Moreover, subsequent update operations will generally require fewer nodes to be split during the invariant-restoration phase, because there will be more empty extra fields available.

The traversal is shown in Algorithm *Cleanup* along with helper function *CleanNode*. We describe it in Section 5.1. We prove that it is correct and complete, that is, it removes exactly what the tracing algorithm told it, in Section 5.2. We show that the node-splitting pointer invariants are maintained in Section 5.3.

Algorithm *Cleanup*$(s, live)$
Input:
1. (∗ Remove all access pointers in dead versions ∗)
2. (∗ $headNode$ is a placeholder, not part of the structure ∗)
3. **for** $headNode$ in $families[s]$
4. **do** $node \leftarrow copy[headNode]$
5. $prevNode \leftarrow headNode$
6. **while** $node$ is not **nil**
7. **do** $prevNode \leftarrow$*CleanNode*$(node, prevNode)$
8. $node \leftarrow copy[node]$
9. **if** $copy[headNode]$ is **nil**
10. **then** remove $headNode$ from $families[s]$
11. (∗ Clean out the version list ∗)
12. $versions[s] \leftarrow live$

Algorithm *CleanNode*$(node, prevNode)$
Input: $node$ is the node to be deleted or cleaned, $prevNode$ is its predecessor
1. $v \leftarrow reachedMin[node]$
2. (∗ Is $node$ dead? ∗)
3. **if** $v = \infty$
4. **then** $copy[prevNode] \leftarrow copy[node]$
5. **return** $prevNode$
6. $version[node] \leftarrow v$
7. **for** f in $extraFields[n]$
8. **do** $u \leftarrow reachedMin[f]$
9. **if** $u = \infty$
10. **then** remove f from $node$
11. **else if** $u = v$
12. **then** $target[origFields[node, name[f]]] \leftarrow target[f]$
13. remove f from $node$
14. **else** $version[f] \leftarrow u$
15. **return** $node$

5.1 Algorithm Operation

The traversal shown in *Cleanup* starts with an initial dummy node, or $head$, of each family, and follows copy pointers from there to visit each node once. As we go through the nodes in a family, we keep track of the previous node in case we need to modify its copy pointer to remove a node from the family.

To enable this traversal, we require constant space per node in the ephemeral structure to maintain a list of these head nodes. This list must be updated when an update operation creates a new node, costing constant additional time. Each visit takes constant time, and so the traversal will take time linear in the size of the data structure.

For each node, we first test whether the trace phase considered it reachable in any live version (*CleanNode* line 3). If not, it is spliced out of its family, and its predecesor node becomes the predecessor of the next node under consideration. Once we remove the pointers to the node within the data structure, the node can be garbage collected normally. Assuming that the trace phase produced correct results, we know that these pointers will be removed, because if they were reachable then this node would be too.

We now continue with the case in which the node is reachable. The node's version stamp is increased to the earliest version in which the node is reachable, implicitly increasing the version stamps of the node's original fields as well.

Then we consider each of the node's extra fields in turn. We consider three possible cases:

1. The field is not reachable in a live version, and so it is removed.

2. The field is first reachable in the same version as its containing node, and hence its contents entirely replace those of the original field with the same name.

3. The field is live version reachable, but starting in some version later than that of the node containing it. Thus, its version stamp is increased just as the node's was.

Note the we do not need to consider the case of an extra field reached before the containing node, because reaching a field requires reaching the node containing it.

After these changes, the nodes and their fields will have all been modified to move changes out of dead versions and remove references to dead versions. We then go through the lists of access pointers and remove those corresponding to dead versions. Now that there are no references to dead versions, we can finally remove the dead versions from the version list, having restored the data structure to a consistent state.

5.2 Correctness

To ensure correctness, we examine and classify the changes to the data structure made by our algorithm. We use this information to verify that our algorithm preserves the invariants maintained by a split node data structure. We also show that our algorithm has sufficiently good time and space performance to be used alongside normal garbage collection.

5.2.1 Possible Changes

By inspection, there are three ways that our algorithm can change the data structure:

1. If a node or pointer is unreachable across its entire valid interval, it will be removed.

2. If the valid interval of an object begins with a sequence of versions before that object's first reachable version, those versions will be trimmed from its valid interval.

3. If a node or pointer is followed by a contiguous interval of one or more versions in which its replacement is unreachable, these will be added to its valid interval. This change occurs implicitly as a result of possible changes 1 and 2.

From the above structural changes, we deduce two important invariants on the data structure:

- Live versions in which an object is reachable are never added to or deleted from the valid interval of any node or pointer. This result follows immediately from the restrictions on the possible changes to the data structure.

- Containment of pointer intervals within nodes is preserved. Modifications performed by our algorithm will only modify nodes and the pointers they contain so that the valid intervals of the pointers remain subsets of the valid interval of the containing node. This result is seen by examining each of the possible changes made to the data structure.

  - If (1) requires a node to be deleted, since the pointer interval is a subset of the containing node interval, any contained pointer is deleted as well.
  - Similarly, if (2) implies that the valid interval of the node needs to be trimmed, then either the valid interval of the pointer starts after the end of the shortening, or (2) will imply that the valid interval of the pointer should be trimmed to the same point as the valid interval of the node itself.

– Likewise, if (3) implies that the valid interval of the pointer should be expanded, either the entire new valid interval will be fully contained in the original valid interval of the node, or (3) will imply that the valid interval of the node should be expanded up to the same point that the valid interval of the pointer is expanded to.

REVISED TEXT ENDS HERE

5.2.2 Invariant Preservation

We now show that our algorithm maintains the pointer invariants of the split node structure. To do so, it will be helpful to define the invariants and the changes imposed by our algorithm in terms of sets of versions.

Given a split node structure S, let V be the set of all versions in the version list for S, and let L be the set of all $v \in V$ such that v is live. Define the relation $<$ on V by $v_1 < v_2$ if v_1 comes before v_2 in the version list. Also, $\forall v \in V$, define v^+ to the version immediately following v in the version list.

A node or pointer a's valid interval consists of a contiguous set of versions. Let $[a_b, a_e]$ denote the valid interval of a, where a_b and a_e are the first and last versions for which it is valid, respectively. Then, from the discussion of Section 5.2.1, applying our algorithm to a results in one of the following:

- If a is not reachable in any live version, it will be deleted.
- Otherwise, a will be processed to yield a', where:

$$\begin{aligned} {a'}_b &= \min\{v \in L | v \geq a_b\} \\ {a'}_e &= \min\{v \in V | v \geq a_e \wedge (v^+ \in L \vee v = \max(V))\} \end{aligned}$$

The Driscoll-Tarjan invariants can be defined in terms of our previous set of definitions. If there is a pointer from p from x to y:

- Proper: p is proper if $p_b \in [y_b, y_e]$
- Nonoverlapping: p is nonoverlapping if $[p_b, p_e] \subseteq [y_b, y_e]$
- Symmetric: for each $i \in [p_b, p_e]$, there exists a pointer q from some node in the family of y to some node in the family of x such that $i \in [q_b, q_e]$.

Similarly, if there is an access pointer a with version i that indicates a node x, the Driscoll-Tarjan invariants can be defined as:

- Proper: a is proper if $i \in [x_b, x_e]$
- Nonoverlapping: a is nonoverlapping if $i \in [x_b, x_e]$

- Symmetric: some node in the family of x contains the label a with version stamp i.

From these definitions, it can be shown that if a pointer is nonoverlapping, it is proper as well. Our proof will focus on demonstrating that pointers are nonoverlapping after our algorithm has completed.

Consider applying our algorithm to S to yield S'. Let p' be a pointer from x' to y' in S'. Then there exists a pointer p from x to y in S, where applying our algorithm to x, y, and p will yield x', y', and p', respectively.

- Nonoverlapping:
  Assume that p is nonoverlapping. Then we have that

$$[p_b, p_e] \subseteq [y_b, y_e]$$

  First, we show that ${y'}_b \leq {p'}_b$. Let

$$\begin{aligned} L_1 &= \{v \in L | v \geq p_b\} \\ \text{and } L_2 &= \{v \in L | v \geq y_b\} \end{aligned}$$

  Then

$$\begin{aligned} {p'}_b &= \min L_1 \\ \text{and } {y'}_b &= \min L_2 \end{aligned}$$

  And thus we have that

$$\begin{aligned} [p_b, p_e] \subseteq [y_b, y_e] &\Rightarrow y_b < p_b \\ &\Rightarrow L_1 \subseteq L_2 \\ &\Rightarrow {y'}_b \leq {p'}_b \end{aligned}$$

  Now, we show that ${p'}_e \leq {y'}_e$. Let

$$\begin{aligned} L_1 &= \{v \in V | v \geq p_e \wedge (v^+ \in L \vee v = \max(V))\} \\ \text{and } L_2 &= \{v \in V | v \geq y_e \wedge (v^+ \in L \vee v = \max(V))\} \end{aligned}$$

  Then

$$\begin{aligned} {p'}_e &= \min L_1 \\ \text{and } {y'}_e &= \min L_2 \end{aligned}$$

  And thus we have that

$$\begin{aligned} [p_b, p_e] \subseteq [y_b, y_e] &\Rightarrow p_e \leq y_e \\ &\Rightarrow L_2 \subseteq L_1 \\ &\Rightarrow {p'}_e \leq {y'}_e \end{aligned}$$

  Together, these give us that

$$\begin{aligned} {y'}_b \leq {p'}_b \leq {p'}_e \leq {y'}_e &\Rightarrow [{p'}_b, {p'}_e] \subseteq [{y'}_b, {y'}_e] \\ &\Rightarrow p' \text{ is nonoverlapping} \end{aligned}$$

- Proper:
  Assume p' is nonoverlapping. Then

$$\begin{aligned}[p'_b, p'_e] \subseteq [y'_b, y'_e] &\Rightarrow y'_b \leq p'_b \leq y'_e \\ &\Rightarrow p'_b \in [y'_b, y'_e] \\ &\Rightarrow p' \text{ is proper}\end{aligned}$$

- Symmetric:
  Let $i \in [p'_b, p'_e]$. Let $j = \max\{v \in L | v \leq i\}$. Since $p'_b \in L$ and $p'_b \leq i$, we have that $j \in [p'_b, p'_e]$. Our algorithm never causes a live version to be added to or deleted from the valid interval of a node or pointer. Thus, $j \in [p_b, p_e]$ as well. Since the symmetric property holds for S, there exists a pointer q in S from some node in the family of y to some node in the family of x such that $j \in [q_b, q_e]$, which implies that our algorithm maps q to some q' in S' such that $j \in [q'_b, q'_e]$, by the same reasoning as above. Furthermore, it is clear that the source and destination nodes for q' will be in the familes of y' and x', respectively. Since $j = \max\{v \in L | v \leq i\}$, we know that

$$\begin{aligned}&\forall v \in V \text{ such that } j < v \leq i, \\ &v \notin \{v \in V | v \geq q_e \wedge (v^+ \in L \vee v = \max(V))\}\end{aligned}$$

  This implies that $i \leq q'_e$, by definition of q'_e. Furthermore, $i \geq j \geq q'_b$, so $i \in [q'_b, q'_e]$. Thus, the symmetry property holds in S'.

We can similarly see that the required invariants are maintained for access pointers. Consider access pointers as pointers whose intervals are only one version long. Then the general proofs demonstrating that any pointer starting in a nonoverlapping and proper state remains that way following our algorithm are immediately applicable. We now show that the symmetry property is maintained as well.

- Symmetric:
  Let a be an access pointer for S. Assume the access array for a now contains a pointer to some node z in position i in S'. Since this pointer wasn't nulled by our algorithm, we know that version i is live. By the symmetry property of access pointers held in S, we know that some node in the family of z contained the label a with version stamp i in S; Since our algorithm never removes live versions from the valid interval of a node or pointer, and never adds versions to the front of an interval, we know this label will still exist, and have version stamp i in S'. Thus, symmetry of access pointers holds in S'.

Step	Time	Space overhead	Complete?
NextLive	$\Theta(v)$	$\Theta(v)$	—
Tracing			
DV	$\Theta(1)$	$\Theta(1)$	No
NaïveTracer	$\Theta(nv_l)$	$\Theta(n)$	Yes
IncrementalTracer	$\Theta(n)$	$\Theta(n)$	Yes, after v_l runs
FastFullTracer	$\Theta(n \log v_l)$	$\Theta(n)$	Yes
Cleanup	$\Theta(n)$	$\Theta(1)$	—

Table 2: Algorithms summary.

5.3 Node-Splitting Invariant Maintenance

6 Proof of Concept Implementation

To test our algorithms, we created a simplified Python implementation of the node-splitting method for creating persistent data structures, using floating point numbers as version stamps. Implementation time bounds have been checked via scaling tests on the number of calls to elementary functions. Space bounds have been verified by similar scaling tests on the number of nodes present. We then implemented our algorithms for garbage collecting and verified that they do perform correctly on persistent deques and binary search trees created with the node-splitting method.

Source code for our implementation of the node-splitting method, several split node data structures, and our algorithms can be found at `http://www.cs.hmc.edu/somewhere/with/code/`.

7 Conclusion

As is often the case with persistent data structures, the creators of the node-splitting method for generalizing persistent data structures do not address the issue of removing some versions of the ephemeral structure from their persistent data structures. Yet split node data structures are a case in which the usual pointer reachability heuristic for liveness does not suffice for collecting memory occupied by unused and inaccessible versions. In order to locate this garbage, a garbage collector must not only examine the syntax of the data structure and its pointer topology, but must also be aware of the semantics of the data structure and its access logic. This is a case in which a specialized method for determining garbage is needed.

The node-splitting method is a powerful tool for creating persistent data structures from a broad class of ephemeral data structures, and with useful asymptotic bounds on performance. However, garbage collection of these data structures is undiscussed in the literature. Our

algorithms collect some or all of the inaccessible nodes. They do this within bounds that trade time for amount of space freed. In doing so, they use no more than a constant factor more space.

However, there are still areas in which our algorithms could be improved. One is the previously mentioned issue of cyclic dependencies. If otherwise unreachable versions are stored in the persistent data structure, or if otherwise unreachable versions can be reached only through a pointer stored in the persistent data structure, then they will not be discovered as live until after our algorithm has run. This necessitates running our algorithm multiple times when new versions are discovered, which could at worst yield $O(n^2)$ performance. However, it does not increase the space costs of the algorithm. If copying collection is used, our algorithm can be rerun on a new copy. If mark-sweep or another similar garbage collection algorithm is being used, we can take advantage of the fact that our algorithm examines each node individually to perform a marking traversal of the structure while ignoring pointers and nodes which would be deleted. Once the final set of accessible, live versions has been determined, then the final, destructive run of the algorithm can be made.

Another limitation of our algorithms are that, although we can collect all garbage, we cannot do so in the optimal $O(n)$ time bound. An open question is whether or not there exists an inexpensive method for collecting all garbage in one collection cycle. Future work thus includes creating such a algorithm or proving that one does not exist.

Despite these limitations, our algorithms still can remove an arbitrary amount of garbage in a fashion not addressed in the published literature. Hopefully these algorithms can expand the usefulness of these persistent data structures into new domains.

8 Related Work

References

Hans-J. Boehm. Destructors, finalizers, and synchronization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–272, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-628-5. doi: http://doi.acm.org/10.1145/604131.604153.

P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 365–372, New York, NY, USA, 1987. ACM Press. ISBN 0-89791-221-7. doi: http://doi.acm.org/10.1145/28395.28434.

James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System*

Sciences, 38(1):86–124, 1989. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/0022-0000(89)90034-2.

James R. Driscoll, Daniel D. K. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. In *SODA '91: Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, pages 89–99, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics. ISBN 0-89791-376-0.

Laura Effinger-Dean, Chris Erickson, Melissa O'Neill, and Darren Strash. Extending garbage collection to complex data structures. In *SPACE 2006*, Charleston, South Carolina, USA, 14 January 2006a.

Laura Effinger-Dean, Chris Erickson, Melissa O'Neill, and Darren Strash. Garbage collection for trailer arrays. In *SPACE 2006*, Charleston, South Carolina, USA, 14 January 2006b.

Martin Erwig. Fully persistent graphs — which one to choose? *Lecture Notes in Computer Science*, 1467:123–??, 1998. URL `citeseer.ist.psu.edu/103974.html`.

Prosenjit Gupta, Ravi Janardan, and Michiel Smid. Efficient algorithms for generalized intersection searching on non-iso-oriented objects. In *SCG '94: Proceedings of the tenth annual symposium on Computational geometry*, pages 369–378, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-648-4. doi: http://doi.acm.org/10.1145/177424.178096.

Chris Okasaki. *Purely functional data structures*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-63124-6.

M.H. Overmars. Searching in the past I. Technical Report RUU-CS-81-07, Institute of Information and Computing Sciences, Utrecht University, 1981a.

M.H. Overmars. Searching in the past II- general transformations. Technical Report RUU-CS-81-09, Institute of Information and Computing Sciences, Utrecht University, 1981b.

Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, 5(3):449–477, 1983. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/2166.357218.

Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, July 1986. ISSN 0001-0782. URL `http://www.acm.org/pubs/toc/Abstracts/0001-0782/6151.html`.

Paul R. Wilson. Uniprocessor garbage collection techniques. URL `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`. Submitted to ACM Computing Surveys, 1994.