Lock-Free Concurrent Binomial Heaps

Gavin Lowe*
Department of Computer Science, University of Oxford
gavin.lowe@cs.ox.ac.uk

August 21, 2018

Abstract

We present a linearizable, lock-free concurrent binomial heap. In our experience, a binomial heap is considerably more complex than previously considered concurrent datatypes. The implementation presents a number of challenges. We need to deal with interference when a thread is traversing the heap, searching for the smallest key: our solution is to detect such interference, and restart the traversal. We must avoid interference between updating operations: we add labels to nodes to prevent interfering updates to those nodes; and we add labels to heaps to prevent union operations interfering with other operations on the same heap. This labelling blocks other operations: to achieve lock-freedom, those threads help with the blocking operation; this requires care to ensure correctness, and to avoid cycles of helping that would lead to deadlock. We use a number of techniques to ensure decent efficiency. The complexity of the implementation adds to the difficulty of the proofs of linearizability and lock-freedom: we present each proof in a modular way, proving results about each operation individually, and then combining them to give the desired results. We hope some of these techniques can be applied elsewhere.

Keywords: Concurrent datatypes, binomial heaps, linearizability, lock freedom.

1 Introduction

A *binomial heap* is a datatype that stores a multiset of integer *keys*. It has operations as declared to the right (we use Scala notation). The effect of the operations is as follows.

```
class BinomialHeap{
  def insert(x: Int): Unit
  def minimum: Option[Int]
  def deleteMin: Option[Int]
  def union(giver: BinomialHeap): Unit
}
```

- The operation insert inserts the given value into the heap.
- The operation minimum returns the minimum key in the binomial heap; in order to deal with an empty heap, it returns an Option[Int] value: either a value Some(x) where x is the minimum key, or None if the heap is empty.

*This is an extended version of a paper under review for the *Journal of Logical and Algebraic Methods in Programming*.

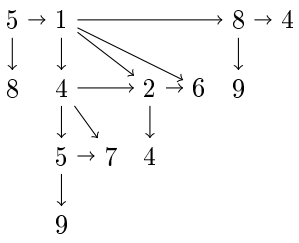
- The operation deleteMin deletes and returns the minimum key (again, using an Option[Int] value).
- The operation union removes all keys from the argument heap ("giver") and adds them to the current heap. We call the former heap the *giving heap*, and the latter the *receiving heap*.

In this paper we describe an implementation of a concurrent binomial heap. Our implementation is linearizable and lock-free (see below).

We start by reviewing *sequential* binomial heaps. Binomial heaps were introduced by Vuillemin in [Vui78]; Cormen et al. [CLR99] give a good description.

A *binomial tree* is composed of *nodes*. Each node n holds an integer *key*, denoted $n.\mathsf{key}$. A binomial tree of degree 0 comprises a single node. A binomial tree of degree $d > 0$ has a root node, and d subtrees with degrees $d-1$, $d-2$, ..., 1, respectively; the root's key is no larger than any other key in the tree. Given two trees t_1 and t_2, each of degree d, with $t_1.\mathsf{key} \leq t_2.\mathsf{key}$, we can merge them into a tree of degree $d+1$ by grafting t_2 onto t_1 as its first child. A binomial tree of degree d has 2^d nodes.

A *binomial heap* is a list of zero or more binomial trees, with their roots linked together; we call this list the *root list*. Each node has a reference next to its next sibling or the next node in the root list, and a list children of references to its children; for convenience, in our concurrent implementation each node also has a reference parent to its parent. The figure to the right illustrates a binomial heap containing four trees, with degrees 1, 3, 1 and 0. (Each next reference is drawn rightwards; references to children are drawn downwards; parent references are omitted.)



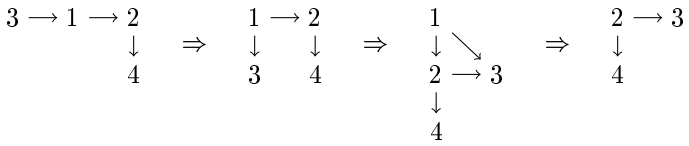
The sequential implementation of a binomial heap ensures the trees in a heap have different degrees, by merging two trees of the same degree, if necessary. This means that a heap with k keys has at most $\lfloor \log k \rfloor + 1$ trees, thereby providing $O(\log k)$ running times for each of the operations. Our concurrent implementation does not rigidly enforce this property, but we aim towards it, in the interests of efficiency. (The sequential implementation also keeps the list in increasing order of degrees; we make no attempt to achieve this property.)

We now sketch how the operations are implemented in a concurrent setting. The minimum operation scans the root list for the node with the minimum key that was not deleted when encountered. Then, if that node has not been deleted in the meantime, it returns the key; otherwise, it restarts. The deleteMin operation again scans the root list for the node with the minimum key. It then attempts to mark that node, thereby indicating that it has claimed the node for deletion. If successful, it then removes that node from the root list, inserting its children (if any) in its place. If the deletion is unsuccessful, it restarts. The insert operation for a key k traverses the root list until it finds either a root of degree 0 with a key that is no larger than k, or the last root node. In the former case, it tries to insert a new node below the singleton root; in the latter case, it tries to insert a new node after the last root. In each case, if it is unsuccessful it retries. The union operation similarly tries to append the first root of the

giving heap after the last root of the receiving heap. We provide two implementations of union: the former makes the simplifying assumption that there are no concurrent operations on the giving heap (although concurrent operations are allowed on the receiving heap); the latter removes this assumption, at the cost of a much more complex implementation.

In addition, threads may attempt to merge two trees into a single tree, via a non-public operation merge; this shortens the root list, thereby making other operations more efficient. The merge operation takes two trees a and b that are expected[1] to have the same degree with a.key $\leq$ b.key, and a node pred that is expected to be the predecessor of b. It attempts to merge a and b into a single tree, making b a child of a (so a is *above* and b is *below*). For the moment, we assume that threads can call this operation at any time, nondeterministically. In Section 10 we explain our tactic for using it.

There are several issues that make the implementation difficult. First, suppose one thread is traversing the root list, searching for the minimum key. Meanwhile, other threads could rearrange the heap concurrently; this can cause the traversing thread to miss nodes in the heap. For example, suppose a traversing thread has reached the 3 in the left-most heap below, and then is suspended. Then suppose other threads merge the 3 below the 1; merge the 2 below the 1; and then delete the 1, reaching the right-most heap below.



Now, if the traversing thread resumes, it detects that it has reached the end of the root list, and incorrectly returns 3. (This problem arises even if the other operations are performed atomically.) Our solution is to detect when the root list has been sufficiently perturbed, and restart the traversal.

Another major difficulty, of course, is that most updates to the heap involve updating more than one node in the heap. For example, merging trees a and b, making the latter a child of the former, involves updating both these nodes and the predecessor pred of b, so involves up to three nodes (it is possible that a = pred). Deleting a node involves updating the node itself (to mark it as deleted), its predecessor, and each of its children (to clear their parent references). Hence such updates cannot be made atomically and efficiently on standard architectures. This leads to intermediate states that are different from ones that can arise when the operations are atomic.

Our approach to merging is based on the technique of Barnes [Bar93]. We start by labelling relevant nodes. Those labels are removed when the operation is completed, or if the operation is unable to complete and has to be backtracked. The labels indicate the intended update to other threads. Informally, this locks those nodes, preventing other threads from interfering with the operation; however, other threads are allowed to update the node to help to complete or backtrack the operation; we use the word "locks" in this sense below. We previously used a similar technique for deletion, labelling both the node to be

[1] We use the word "expected" to indicate conditions that the thread in question should check before the operation is called, but that subsequently might have been invalidated.

deleted and its predecessor. However, we found that it was more efficient to label just the node to be deleted: this might mean that the predecessor subsequently changes, and we have to find the new predecessor, but the benefits outweigh this cost.

Extra difficulties arise if we allow operations concurrent to a giving union (i.e. where the heap is the giving heap in a union). The union may interfere with other operations, for example causing an insertion to actually insert the new node in the receiving heap, or for a minimum or deleteMin to return a key from the receiving heap. Our approach to avoiding such problems extends the above labelling technique. We add a label to the head node of each heap during a union; likewise, we add a label to the head node for the critical step of insertions and deletions: this ensures that these operations cannot happen concurrently to a giving union. We also need efficient techniques to detect if a giving union has happened during some period, and to find the current heap containing a node.

Finally, we aim to make the data structure lock-free: if threads (collectively) are scheduled sufficiently often, then eventually some operation completes. This means that if a thread is blocked by the labelling of nodes described in the previous paragraph, that thread should help to complete the operation. Ensuring that no thread can get stuck in a loop requires great care.

Our approach to proving lock-freedom is as follows. For most operations, each update to a node represents positive progress: we can bound the number of updates that an operation makes, excluding helping. Thus, in any execution that represents a failure of lock-freedom, such updates must eventually end. The exception to this is with merging; we use a different mechanism to bound the number of such updates done by merges. Thus, it is enough to prove that each operation terminates in a finite number of steps *assuming* no other thread updates a node. We do this by showing three things (under this assumption):

- That each operation terminates in a finite number of steps, other than, perhaps, helping another operation;
- Likewise that each attempt to help terminates in a finite number of steps, other than, perhaps, recursively helping another operation;
- That each chain of recursive helping is finite (in particular, acyclic).

In our experience, a binomial heap is considerably more complex than previous concurrent datatypes. Part of our goal in undertaking this work was to understand the degree to which concurrent versions of more complex datatypes are possible, and how best to reason about them. (Our proofs are rigorous but not machine-checked; we leave a machine-checked proof as a challenge.)

We consider our main contributions to be the combination of techniques that we used to overcome the above challenges and to improve the efficiency of the implementation, together with the verification. In addition, we consider the binomial heap potentially useful in its own right: in several use cases, our implementation out-performs other implementations of concurrent priority queues (i.e. omitting the union operation from the interface); further, we are not aware of other lock-free implementations giving the same interface (including union).

During the development, we intensively tested the implementation for linearizability [HW90] using the techniques from [Low17]. This revealed several subtle errors with earlier versions of the implementation. It is our experience

that this technique is very effective at discovering errors, if they exist; conversely, not finding errors improves our confidence, and justifies the effort of a formal proof.

Our implementation, in Scala, is available from `http://www.cs.ox.ac.uk/people/gavin.lowe/BinomialHeap/index.html`. We present the design below in pseudo-Scala (taking a few small liberties with syntax).

The implementation of each operation is, to a certain extent, dependent on the others; we describe the operations in a slightly unnatural order, in order to produce a coherent explanation. In particular, the implementation of functions that traverse the root list depend on the implementation of other operations. In the next section we define the types of nodes, and give some basic definitions. In Section 3 we describe how to merge two trees. In Section 4 we describe how to delete a node, assuming the minimum node and its expected predecessor have been found; we also assume a function to find the new predecessor of the node being deleted, if it has changed in the meantime; we present that latter function in Section 5, where we also present our technique for safely traversing the root list. In Section 6 we describe the insert operation. In Sections 7 we describe the union operation under the assumption that there are no concurrent operations on the giving heap. In Sections 8 and 9 we describe the minimum and deleteMin functions. We describe our tactic for when we attempt to merge trees in Section 10. In Section 11 we present a revised version of union, where we relax the assumption about no concurrent operations on the giving heap; this also requires making some small changes to the previous operations, which we describe in Section 12. We prove that the implementation is linearizable in Section 13, and we prove lock-freedom in Section 14. We sum up in Section 15. In places, in the interests of clarity, we present slightly simplified versions of the implementation, and then sketch enhancements without giving full details.

Linearizability Our correctness condition is the well-established and accepted condition of *linearizability* [HW90]. Informally, a concurrent datatype C is linearizable with respect to a sequential (specification) datatype S if every history c of C is linearizable to a history s of S: each operation op of c appears to take place atomically at some point in time, known as the *linearization point*, between the invocation and return of op; and the operations, ordered according to their linearization points, form a legal history of the sequential datatype S (i.e. respects the intended sequential semantics). See [HW90, HS12] for a formal definition; but this definition suffices for our purposes.

Related work To our knowledge, the only prior implementation of a concurrent binomial heap is by Crupi et al. [CDP96]. However, their setting is very different from ours: a *single* operation takes place at a time, and *all* threads cooperate on each operation. They show each of the main operations can be performed on a heap containing n keys in time $O(\log\log n + \frac{\log n}{p})$ with $p = O(\frac{\log n}{\log\log n})$ processors.

Huang and Weihl [HW91] describe a lock-based priority queue based on Fibonacci heaps, where deleteMin operations are non-strict, i.e. they may return values that are not necessarily minimal.

In addition, various authors have given implementations of priority queues (without the union operation) based on skiplists. Lotan and Shavit [LS00] de-

scribe a design that is not linearizable, since a deleteMin operation may ignore a smaller value that is inserted concurrently. Further, the design uses locks, so is clearly not lock-free. Sundell and Tsigas [ST05] describe a lock-free priority queue based on a skiplist. The design also turns out not to be linearizable[2]; but this is easily fixed. Lindén and Jonsson [LJ13] describe a linearizable skiplist-based implementation of a priority queue, that uses various clever techniques to minimise the number of update operations necessary for a deleteMin operation, and so provide good performance.

Liu and Spear [LS12] describe an implementation of a priority queue using a mound: a binary heap where each node holds a linked list sorted by priorities; insertion runs in $O(\log(\log(N)))$ and deleteMin in $O(\log(N))$.

Braginsky et al. [BCP16] present an implementation of a priority queue based upon a linked list of *chunks*, where a chunk uses an array to store some items of data. Each chunk holds data that are smaller than those in the following chunk; but only the data in the first chunk (on which deleteMin operates) are kept sorted. When a chunk becomes full, it is split into two. A skip list allows fast access to the appropriate chunk. A form of elimination between concurrent insert and deleteMin calls is implemented.

Some papers, e.g. [AKLS15], have considered *relaxed* priority queues, where the deleteMin operation is not constrained to return the absolute minimum key; this avoids the minimum key becoming a bottleneck, so gives better performance.

2 Basic types and definitions

In this section we describe the implementation of nodes. Each node has the following fields:

- key: Int, which is immutable;
- parent, next: Node (each possibly null), pointing to the node's parent, and next sibling or next root, respectively;
- children: List[Node], a list of the node's children (possibly empty);
- degree: Int, giving the degree of the node;
- label: Label (possibly null), describing any update currently operating on the node; we introduce the different types of labels when we describe the relevant operations;
- seq: Int, a sequence counter that is updated each time a node is moved from being a root to become a child of another node, or when the node is labelled for deletion; we use this when traversing the root list, to detect changes in the root list that require the traversal to restart (see Section 5);
- deleted: Boolean, a flag that is set when a node has been fully deleted; we explain this more when we discuss deletion in Section 4.

[2] Acknowledged by the authors (personal communication).

```
abstract class Label
class NodeState(val parent: Node, val degree: Int, val children: List[Node],
                val next: Node, val seq: Int, val label: Label){
  def unlabelled = label == null
  def parentless = parent == null
}
class Node(val key: Int){
  /** The state of this node, atomically updateable. */
  private val state = new AtomicReference[NodeState](new NodeState(
    parent = null, degree = 0, children = List(), next = null, seq = 0, label = null))
  /** Get the current state. */
  def getState : NodeState = state.get
  /** CAS the current state. */
  def compareAndSet(oldState: NodeState, newState: NodeState) : Boolean =
    state.get == oldState && state.compareAndSet(oldState, newState)
  /** Has this Node been fully deleted? */
  @volatile var deleted = false
  /** Get the current sequence number. */
  def getSeq = getState.seq
  /** Get the current label. */
  def getLabel = getState.label
}
```

Figure 1: NodeState and Node

To update several parts of a node's state *atomically*, we encapsulate most of the mutable state into a NodeState object (with immutable fields); see Figure 1. We use suggestive notation for defining new NodeState objects: we write, for example, state[next ↦ n] for a NodeState that is the same as state except with its next field set to point to n. We include in NodeState functions that test if the label or parent is null.

A Node itself comprises the (immutable) key, a reference state to the state, and the deleted field[3] (we do not include deleted within the NodeState, because we never need to update it at the same time as another field, nor in a way that depends on any other field). Each Node includes operations to (atomically) read the state, perform a compare-and-set (CAS)[4] on the state, and to get the sequence number or label. In our informal commentary, we sometimes write, e.g., n.next as shorthand for n.getState.next.

The binary heap is implemented as a linked list of root nodes, linked via their next fields; we call this list the *root list*. For convenience, we use a dummy header node head (with an arbitrary key).

```
private val head = new Node(−1)
```

For simplicity, we assume a garbage collector. Whenever we update the state of a node, we do so with a *new* object; this avoids the ABA problem [HS12].

We state some invariants of the binomial heap. The following invariants

[3] The class java.util.concurrent.atomic.AtomicReference provides references to objects which can be updated atomically, using compare-and-set operations, and that provide sequentially consistent memory accesses. The @volatile annotation provides sequentially consistent memory accesses to that field.

[4] compareAndSet(oldState, newState) behaves as follows: if state equals oldState, then it is updated to newState, and true is returned; otherwise, state is unchanged, and false is returned. We check that the state equals oldState before attempting the CAS operation: this is standard practice, to avoid creating lots of memory bus traffic via a CAS that is bound to fail.

capture the standard property of keys in a binomial tree, and some other obvious properties.

Invariant 1 *If n_2.parent $= n_1$ and $n_1 \neq$ null then n_1.key $\leq n_2$.key.*

Recall that we use a Delete label to indicate that a node is being deleted.

Invariant 2 *For each node n that does not have a Delete label: n.children contains n.degree nodes, which are joined by next pointers in a linked list; and c.parent $= n$ for each $c \in n$.children.*

Invariant 3 *The parent references are acyclic.*

Labels are added only to root nodes, except a Delete label may be added to a node whose parent is being deleted.

Invariant 4 *If a node has a non-null parent, then either it has a null label, or both it and its parent have Delete labels.*

(We state further invariants later.)

We give rules under which operations may change nodes.

Rule 1 *If a node n has a non-null parent, the only changes that may be performed on n are to help with the deletion of the parent, or to label n for deletion if its parent is being deleted; in each case the parent will have a Delete label.*

We make this more precise in Section 4 when we describe deletion. Our other rules concern nodes with labels, restricting the updates that may be performed on the node, often to just updates that help to complete the corresponding operation, or to remove the label if the operation is backtracked. We give these rules when we present the corresponding labels. When we analyse an operation, we do so under the assumption that all other operations follow these rules. We verify that this is indeed the case when we analyse those other operations.

3 Merging trees

The merge function merges two trees a and b, so that a ends up *above*, and b *below*. More precisely, it takes three trees, a, b and pred (where possibly a = pred, but otherwise the trees are distinct), and their expected states aState, bState and predState. It assumes that a.key $\leq$ b.key, predState.next = b, aState and bState have the same degrees, and all are unlabelled roots with null parents. It attempts to merge a and b, making a the parent of b. Recall that for the moment we assume that threads can call merge at any time, nondeterministically, subject to the above precondition; in Section 10, we explain our tactic for using it.

The merge function is a private function: it is done only to optimise performance by reducing the length of the root list. We therefore don't care much if a call to merge is unsuccessful: if another operation interferes with it, it will detect that the state of a node has changed (which might invalidate the conditions for the merge), backtrack by undoing its previous steps, and give up; however, we do ensure that it terminates in a finite number of steps.

We start by sketching how the operation proceeds, and then flesh out the details, below. Suppose, for the moment, a $\neq$ pred. The merge function proceeds as follows; the steps are illustrated in Figure 2.

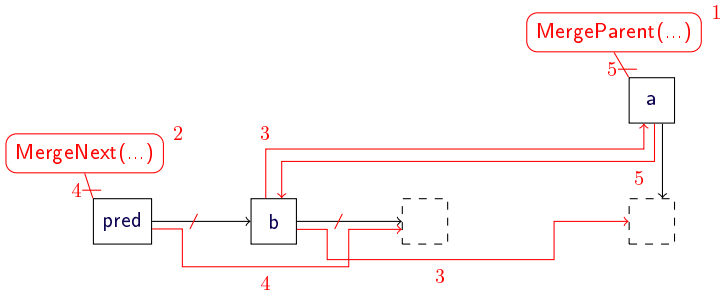


Figure 2: Illustration of the merge function. next references are drawn to the right, parent references upwards, and references to the first child downwards; dashed boxes illustrate possibly null nodes; updates are indicated in red.

1. It tries to add a MergeParent label to a. We call this the a-*labelling step*. If the state of a has changed, this fails, and merge gives up.
2. It tries to add a MergeNext label to pred. We call this the pred-*labelling step*. If this fails (and no other helping thread has added the MergeNext label), it backtracks, removes the label from a, and gives up.
3. It tries to update b, atomically setting its parent field to a, and its next field to a's first child, or to null if a has no children. We call this the b-*update step*. If this fails (and no other thread has done the update), it backtracks, removing the labels from pred and a, and gives up. If this update succeeds, the merge is guaranteed to complete successfully.
4. It tries to update pred, atomically setting its next field to bState.next, and removing the label. We call this the pred-*update step*.
5. It tries to update a, atomically incrementing its degree, adding b to its children, and removing the label. We call this the a-*update step*.

All steps except for the a-labelling may be performed by other helping threads. Note that if one of the labelling steps fails, because the state of the relevant node has changed, then the merge gives up: in most such cases, it is not possible to complete the merge, because the relevant parts of the precondition no longer hold. Note that it is fine to just give up, since merging is just an optimisation.

If a = pred, things are slightly simpler. The pred-labelling can be omitted, and subsumed into the a-labelling. Similarly, the pred-update and a-update can be combined.

The two labels contain enough information to allow helping threads to complete the operation; they are defined as follows.

```
case class MergeParent(pred: Node, predState: NodeState, pLabel: MergeNext,
                       b: Node, bState: NodeState) extends Label
case class MergeNext(a: Node, b: Node, bState: NodeState) extends Label
```

Within MergeParent, if a = pred the pLabel parameter is bound to the corresponding MergeNext label for pred; if a = pred, it is bound to null; each other parameter is bound to the corresponding parameter of merge. The following invariant captures properties of the relevant nodes.

```
1  private def merge(a: Node, aState: NodeState, pred: Node, pState: NodeState,
2      b: Node, bState: NodeState) = {
3    assert(a != b && a.key <= b.key && aState.degree == bState.degree &&
4             pState.next == b && aState.unlabelled && aState.parentless &&
5             bState.unlabelled && bState.parentless && pState.unlabelled && pState.parentless)
6    val pLabel = if(a != pred) MergeNext(a, b, bState) else null
7    val aStateL = aState[label ↦ MergeParent(pred, pState, pLabel, b, bState)]
8    if(pred.getState == pState && b.getState == bState &&
9         a.compareAndSet(aState, aStateL))
10     if(a != pred) mergeLabelPred(a, aStateL, pred, pState, pLabel, b, bState)
11     else mergeUpdateB(a, aStateL, pred, pState, b, bState)
12 }
13 private def mergeLabelPred(a: Node, aStateL: NodeState, pred: Node,
14     pState: NodeState, pLabel: MergeNext, b: Node, bState: NodeState) = {
15   assert(a != pred && aStateL.label.isInstanceOf[MergeParent])
16   val pStateL = pState[label ↦ pLabel]
17   if(b.getState == bState && pred.compareAndSet(pState, pStateL))
18     mergeUpdateB(a, aStateL, pred, pStateL, b, bState)
19   else if((pred.getState.label ne pLabel) && b.getState.parent != a)
20     a.compareAndSet(aStateL, aStateL[label ↦ null]) // backtrack
21 }
22 private def mergeUpdateB(a: Node, aStateL: NodeState, pred: Node, pStateL: NodeState,
23     b: Node, bState: NodeState) = {
24   assert(aStateL.label.isInstanceOf[MergeParent] &&
25            (a != pred || pStateL.label.isInstanceOf[MergeNext]))
26   val newNext = if(aStateL.children.isEmpty) null else aStateL.children.head
27   val bStateU = bState[parent ↦ a, next ↦ newNext, seq ↦ bState.seq+1]
28   if(b.compareAndSet(bState, bStateU))
29     if(a != pred) mergeUpdatePred(a, aStateL, pred, pStateL, b, bState.next)
30     else mergeUpdateAPred(a, aStateL, b, bState.next)
31   else if(b.getState.parent != a){ // backtrack
32     if(a != pred) pred.compareAndSet(pStateL, pStateL[label ↦ null])
33     a.compareAndSet(aStateL, aStateL[label ↦ null])
34 } }
```

Figure 3: The merge, mergeLabelPred and mergeUpdateB functions.

Invariant 5 *If node a has a label $\mathit{MergeParent}(pred, pState, pLabel, b, bState)$, then*

$$a \neq b \wedge a.\mathit{key} \leq b.\mathit{key} \wedge a.\mathit{degree} = bState.\mathit{degree} \wedge a.\mathit{parentless} \wedge pState.\mathit{unlabelled} \wedge pState.\mathit{parentless} \wedge pState.\mathit{next} = b \wedge (a = pred \Rightarrow a.\mathit{next} = b).$$

If node p has a label $pLabel = \mathit{MergeNext}(a, b, bState)$, then a has a label $\mathit{MergeParent}(p, p.\mathit{getState}[\mathit{label} \mapsto \mathit{null}], pLabel, b, bState)$. Note that, combined with the above, this implies

$$a \neq b \wedge a.\mathit{key} \leq b.\mathit{key} \wedge a.\mathit{degree} = bState.\mathit{degree} \wedge a.\mathit{parentless} \wedge p.\mathit{parentless} \wedge p.\mathit{next} = b.$$

In more detail, the function merge (Figure 3) attempts the a-labelling. It prepares the two labels and, if the states have not changed, attempts to label a. This locks a and its children. The fact that this thread prepares both labels means that the MergeNext label is uniquely identified with this invocation of merge, and cannot be confused with another label with identical arguments.[5]

[5]We use "eq" and "ne" for equality and inequality tests between non-null labels; these

```
1  private def mergeUpdatePred(a: Node, aStateL: NodeState, pred: Node, pStateL: NodeState,
2      b: Node, nextNode: Node) = {
3    assert(a != pred && aStateL.label.isInstanceOf[MergeParent] &&
4             pStateL.label.isInstanceOf[MergeNext] &&
5             (b.getState.parent == a || a.getState != aStateL && pred.getState != pStateL))
6    val pStateU = pStateL[next ↦ nextNode, label ↦ null]
7    if(pred.compareAndSet(pStateL, pStateU)) mergeUpdateA(a, aStateL, pred, b)
8  }
9  private def mergeUpdateA(a: Node, aStateL: NodeState, pred: Node, b: Node) = {
10   val aStateU =
11     aStateL[degree ↦ aStateL.degree+1, children ↦ b::aStateL.children, label ↦ null]
12   a.compareAndSet(aStateL, aStateU)
13 }
14 private def mergeUpdateAPred(a: Node, aStateL: NodeState, b: Node, nextNode: Node) = {
15   val aStateU = aStateL[degree ↦ aStateL.degree+1, children ↦ b::aStateL.children,
16                         next ↦ nextNode, label ↦ null]
17   a.compareAndSet(aStateL, aStateU)
18 }
```

Figure 4: The mergeUpdatePred, mergeUpdateA and mergeUpdateAPred functions.

The function mergeLabelPred attempts the pred-labelling. If successful, this locks pred and its children. If the labelling is unsuccessful, and no other thread has performed this labelling, it backtracks and removes the label from a.

The b-update step is attempted by the function mergeUpdateB. It attempts to update b, setting its parent and next fields appropriately. If this is successful, the merge is guaranteed to complete. If the update is unsuccessful, and no other thread has performed this step, then it attempts to backtrack.

When pred $\neq$ a, the pred update and a updates are performed by the functions mergeUpdatePred and mergeUpdateA (Figure 4), respectively[6]. The case pred = a is handled by mergeUpdateAPred.

The following rule concerns MergeParent and MergeNext labels.

Rule 2 *If a node has a MergeParent label, the only updates allowed on it are to perform the a update, to remove the label if the merge cannot be completed, or to perform the pred update if a = pred. If a node has a MergeNext label, the only updates allowed on it are to perform the pred update, or to remove the label if the merge cannot be completed.*

If a thread performing another operation is unable to proceed because it is blocked by the merge, then it helps the merge; this might be because the other operation needs to update a node that has been labelled by the merge, but is unable to do so because of the above rule. Thus once the initial labelling step has happened, some of the steps may be performed by such helping threads. Any thread helping does so via the help function in Figure 5, which calls into the appropriate sub-function of merge[7] (line 26 concerns helping with deletions; we explain this case in Section 4). We design the code within the merging

correspond to *reference* equality: using "==" and "!=" would correspond to *value* equality (this is the default for case classes), which would be incorrect in some places.

[6] "::" represent cons, adding an element to the front of a list.

[7] The code makes use of Scala pattern matching. The pattern in line 17, using "@", binds the variable pLabel to the label. The pattern in line 20, using back ticks, matches just a MergeParent label whose fields match the given values.

functions to allow such helping: updates on nodes take place only from the expected states.

At various places, a thread may detect that another thread has already performed the next step. For example, in mergeLabelPred, the thread may detect that another thread has already labelled pred (line 19 of Figure 3). In such cases, it allows the thread that performed that step to complete the operation. This is for pragmatic reasons, since it is inefficient for two live threads to be working on the same operation: one of the threads could be doing something else; and if both threads were working on the same operation, then when one thread performs a particular CAS operation, it would invalidate the relevant cache lines of the other thread, forcing extra memory accesses.

3.1 Correctness

We now argue the correctness of the merge operation: it either merges the two trees, or leaves the heap essentially unchanged (maybe just replacing a's and pred's states with equivalent ones).

Lemma 1 *The implementation works correctly when it operates without interference from other threads (assuming the stated precondition).*

Proof: (Sketch.) Each step of the operation succeeds. The final states of the three nodes correspond to the merge having completed. □

We now consider helping by other threads. The following lemma identifies conditions under which threads may help with different stages of a merge.

Lemma 2 *Suppose a thread calls:*

- *mergeLabelPred after the a-labelling CAS with $a \neq pred$;*
- *mergeUpdateB after the pred-labelling CAS, or after the a-labelling CAS with $a = pred$;*
- *mergeUpdatePred after the b update CAS with $a \neq pred$;*
- *mergeUpdateA after the pred update CAS with $a \neq pred$; or*
- *mergeUpdateAPred after the b update CAS with $a = pred$;*

in each case, with the same values of the parameters as the primary thread. Then the behaviour is correct, either contributing to the merge, or not interfering with it.

Proof: Careful examination of the code shows that such helping threads contribute correctly to the deletion: each update on a node is executed only from an appropriate state, so if several threads attempt such an update, only the first will succeed.

Note in particular that the backtracking within mergeLabelPred (Figure 3) is done correctly: it won't be done if another thread has labelled pred, since in that case either: (a) pred will still have the label pLabel (before the pred update), or (b) b's parent will equal a (after the b update until after the a update), or (c) a's state will have changed (after the a update); in each case, the backtrack CAS won't be attempted.

```
1  private def help(helpNode: Node, helpState: NodeState) = helpState.label match{
2    case MergeParent(pred, pState, pLabel, b, bState) =>
3      if(b.getState.parent == helpNode){  // b update has happened
4        val newPState = pred.getState
5        if(helpNode == pred)
6          mergeUpdateAPred(helpNode, helpState, b, bState.next)
7        else if(newPState.label eq pLabel) // pred update has not happened
8          mergeUpdatePred(helpNode, helpState, pred, newPState, b, bState.next)
9        else mergeUpdateA(helpNode, helpState, pred, b)
10     }
11     else{
12       val newPState = pred.getState
13       if(helpNode != pred && (newPState.label ne pLabel))
14         mergeLabelPred(helpNode, helpState, pred, pState, pLabel, b, bState)
15       else mergeUpdateB(helpNode, helpState, pred, newPState, b, bState)
16     }
17   case pLabel @ MergeNext(a, b, bState) =>
18     assert(a != helpNode); val newAState = a.getState
19     newAState.label match{
20       case MergeParent(`helpNode`, _, `pLabel`, `b`, `bState`) =>
21         if(b.getState.parent == a)  // b update has happened
22           mergeUpdatePred(a, newAState, helpNode, helpState, b, bState.next)
23         else mergeUpdateB(a, newAState, helpNode, helpState, b, bState)
24       case _ => { } // a update has happened
25     }
26   case Delete(pred) => helpDelete(pred, helpNode, helpState)
27 }
```

Figure 5: Helping operations.

Also note that the backtracking in mergeUpdateB (Figure 3) is done correctly: it won't be done if another thread has updated b as part of the merge, since in that case either (a) b's parent will equal a (until after the a update), or (b) the state of pred or a will have changed (at the pred update or a update); in each case, the backtrack CAS won't be attempted. □

Any thread helping will do so via the help function in Figure 5.[8] (Line 26 concerns helping with deletions; we explain this case in Section 4.)

Lemma 3 *If a thread helps with the merge via the help function, it does so correctly.*

Proof: Careful examination of the code shows that helping threads call into the correct sub-function, as captured by Lemma 2. (Note that the order in which b's and pred's states are read in lines 3 and 4 is critical: in order to be sure that the pred update has happened at line 9, we need to be sure that pred's state was not read before the pred-labelling step.) □

Now consider interference from other operations.

Lemma 4 *Other operations do not interfere with the correctness of merging.*

Proof: If any thread updates a before the a-labelling step, then the merge fails, and leaves the heap unchanged.

[8] The code makes use of Scala pattern matching. The pattern in line 17, using "@", binds the variable pLabel to the label. The pattern in line 20, using back ticks, matches just a MergeParent label whose fields match the given values.

Once a has been labelled, by Rule 2, no other operation may update it other than to help with the merging, until it is unlabelled in the a update.

If any thread updates pred before the pred-labelling step, then the merge fails, the labelling of a is backtracked, and the heap is left essentially unchanged.

Once pred has been labelled, by Rule 2, no other operation may update it other than to help with the merging, until it is unlabelled in the pred update; subsequent updates do not interfere with the merge.

If any thread updates b before the b update, then the merge fails, the labelling of pred and b are backtracked, and the heap is left essentially unchanged.

After the b update, but before the a update, no thread may update b: by Rule 1, no thread may update it other than as part of the deletion of its parent; but its parent a still has a MergeParent label, so any attempt to delete a is blocked until the merge is complete. □

Lemma 5 *The merge operation maintains Invariants 1, 2, 3 and 4, and follows Rules 1, 2 and later rules concerning nodes with labels (assuming the stated precondition).*

Proof: The precondition includes that a.key $\leq$ b.key, so Invariant 1 is maintained.

The b update adds b to the front of the linked list of a's children, and sets b.parent = a; the a update then updates a's children and degree, maintaining Invariant 2.

Invariant 3 is maintained: the only point at which a parent reference is set to a non-null value is the b update, setting b.parent = a; but a has a null parent, so this does not create a cycle.

Labels are added to nodes only from their initial states; the precondition includes that these states have null parent fields, so Invariant 4 is maintained by the labelling steps. Similarly, b's parent is updated only from b's initial state, and the precondition includes that this state is unlabelled, so Invariant 4 is maintained by the b-update step. The other steps trivially maintain this invariant.

The precondition includes that the three nodes are initially unlabelled roots. Hence any updates from these initial states trivially satisfy the rules. The only other updates are the update and backtrack steps, from the labelled states of a and pred, which satisfy the conditions of Rule 2.

All the updates trivially satisfy Rule 1, since each node has a null parent. It trivially satisfies later rules concerning labels, since it updates no labelled node. □

Lemma 6 *Invariant 5 is satisfied (assuming the stated precondition).*

Proof: The stated precondition shows that most of the clauses of the first part of the invariant are satisfied when the MergeParent label is added. For the final clause "$a = pred \Rightarrow a.\mathsf{next} = b$", note that if a = pred, the a-labelling CAS will happen only if aState = pState, so the result follows since pState.next = b. By Rule 2, no thread will change the state of this node until the label is removed.

The MergeNext label is added after the MergeParent label, and removed before it (both in normal operation and when backtracking). By Rule 2, no other thread may interfere with this. □

We now prove a lemma that will be useful for proving liveness properties.

Lemma 7 *Any call to merge will terminate in a finite number of steps.*

Proof: merge and its subfunctions contain no loop or recursion. □

Lemma 8 *Each thread can call help on a node with a particular MergeNext or MergeParent label at most four times. Each call terminates in a finite number of steps.*

Proof: Essentially, for each call to help, some progress is made, either by this thread, or by some other thread after helpState was read. We consider the sub-function called by help.

- If help calls mergeUpdateA or mergeUpdateAPred, the merge will be completed and a's label removed (either by this thread or another).
- If help calls mergeUpdatePred, the pred update will be performed (either by this thread, or by another thread after the read of newPState in the MergeParent case, or by another thread after the read of helpState in the MergeNext case).
- If help calls mergeUpdateB, the b update will be performed (either by this thread, or by another thread after the relevant read of b's state).
- If help calls mergeLabelPred, the pred labelling will be performed (either by this thread, or by another thread after the read of newPState).
- If no sub-function is called, in the MergeNext case, the merge has been completed since the read of helpState.

If the above does not complete the merge, it might lead to a subsequent call to help by the same thread; but that will call a later sub-function of merge. Hence at most four such calls will complete the merge.

Each call terminates in a finite number of steps, by Lemma 7. □

4 Deleting nodes

In this section we describe how a node is deleted. In Section 9, we will describe how this is invoked by the deleteMin function. We assume here that the node delNode to be deleted and its expected predecessor pred have been identified. Recall that we allow only root nodes to be deleted; however, we allow a thread to claim a non-root node n for deletion if its parent p is being deleted; however, the deletion of p must be completed before continuing with the deletion of n.

The deletion itself is encapsulated in functions delete and deleteWithParent, the latter dealing with the case of a non-root. The attempt to delete delNode may fail, for example if another thread has already claimed it for deletion; each function returns a boolean to indicate if it was successful. If it fails, the deleteMin function either calls the function again, if this could feasibly succeed, or identifies a new node to try to delete.

It is possible that the predecessor of delNode changes before the relevant step of the deletion. In this case, it is necessary to identify the new predecessor. We describe the search for the new predecessor in Section 5. Informal experiments suggest the predecessor is often unchanged, so recording it is worthwhile.

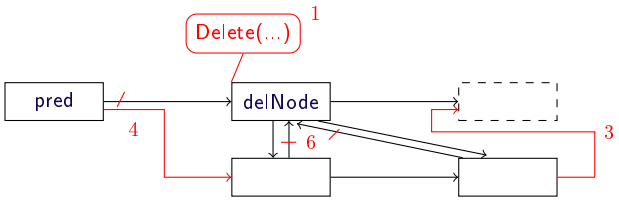


Figure 6: Illustration of deletion. Drawing conventions are as in Figure 2.

Deletion proceeds in several stages. We give a brief overview here, and give more detail below; the steps are illustrated in Figure 6.

1. It attempts to label delNode with a Delete label. We call this the *delNode-labelling step*. If this is successful, the deletion is guaranteed to succeed.
2. If the parent of delNode is being deleted, it helps with this deletion.
3. If delNode has children, the last child is updated to point to the successor of delNode.
4. If pred is still the predecessor of delNode, its next field is updated to point to delNode's first child, if it has one, or otherwise to delNode.next, thereby removing delNode from the root list; note that in the case that delNode has children, delNode.next is now reachable via the last child. If pred is not still the predecessor of delNode, a search is carried out to find the new predecessor, and the above update done to it. We call this the *pred-updating step*.
5. delNode's deleted field is set, to indicate that it has been fully deleted.
6. The parent fields of delNode's children (if any) are cleared.

All steps except step 1 might be carried out by other threads helping.

Part of the rationale for our design is to allow a thread to claim the node it will delete, via the delNode-labelling step, as quickly as possible, so that threads waste as little time as possible in deletion attempts that subsequently fail. In a previous design, threads first labelled the precedessor pred, before labelling delNode (backtracking if the latter failed): this approach meant that the predecessor could not subsequently change, but proved slightly slower in practice. Similarly, we previously did not allow a non-root n to be labelled for deletion; in such cases, all threads performing a deleteMin could identify n as the minimum node, and *all* would help with the deletion of its parent before attempting to label n, which is inefficient; using our current approach, only the thread that succeeds in labelling n helps with its parent.

We say that a node is *marked for deletion* (or just *marked*) if it has a Delete label. We say that it is *decoupled* if the pred update has occurred. We say that it is *fully deleted* if its deleted field is set.

We now make precise the circumstances under which a thread may change the state of a node with a non-null parent.

Rule 1 *(refined) If a node n has a non-null parent p, the only changes that may be performed on n are as follows:*

- *if p is marked for deletion, and n is p's last child, then n may have its next field set to p's successor;*
- *if p is deleted, then n may have its parent field cleared;*
- *to mark n for deletion.*

The Delete label includes a reference to the expected predecessor node. It also includes a timestamp, giving the time that it was created: we use this in the minimum function to determine if the node was unmarked when the function was invoked.

```
case class Delete(pred: Node) extends Label{
  val ts = java.lang.System.nanoTime // time label was created
}
```

Rule 3 *If a node has a Delete label, the only subsequent changes allowed to it are to clear its parent field if its parent is deleted, or to set its deleted field once the node is decoupled. In particular, delNode.next does not change.*

Deletion of a node with a null parent is encapsulated into the delete function (Figure 7). This takes as arguments the node delNode to be deleted, its expected state delState, and its expected predecessor pred. It calls labelForDelete, which tries to update delNode using a CAS, to add a Delete label and increase the sequence number; if successful, it returns the resulting state; in this case, the deletion is bound to succeed.

The function deleteWithParent deals with the case that the parent is non-null. Recall that in this case, all delNode's ancestors, up to the root list, are being deleted. It again calls labelForDelete to mark delNode for deletion; in the case that delNode.next is null, and hence delNode is parent's last child, it also updates delNode.next to parent.next, thereby performing the last child update on parent. If the marking is successful, it helps with the deletion of parent before completing the deletion of delNode.

The deletion is completed via the completeDelete function. If delNode has no children, it uncouples delNode from the previous node by updating the previous node's next field to point to delNode's successor, next, via the function predUpdate (see below). Otherwise, it first tries to update the last child's next pointer to point to the root node after delNode; this update fails only if it has already been performed by another thread. It then uncouples delNode via predUpdate, updating the previous node's next field to point to delNode's first child. Finally, it clears the parent field of each child.

The function predUpdate controls the update of the predecessor of delNode to point to newNext; the actual update and the setting of delNode's deleted field are done by the function completePredUpdate (Figure 8). delNode's deleted field is checked at various places, in case another thread has already completed the deletion. If pred is still the root node before delNode, completePredUpdate is called (line 4) to attempt to decouple delNode. Otherwise, or if the decoupling is unsuccessful, it is necessary to search for the new predecessor. The search is encapsulated in the function findPred. We present findPred in Section 5, and

```
1  private def delete(pred: Node, delNode: Node, delState: NodeState) : Boolean = {
2    assert(delState.unlabelled && delState.parentless)
3    val delStateL = labelForDelete(pred, delNode, delState, delState.next)
4    if(delStateL != null){ completeDelete(pred, delNode, delStateL); true }
5    else false
6  }
7  private def labelForDelete(
8      pred: Node, delNode: Node, delState: NodeState, newNext: Node): NodeState = {
9    val delStateL = delState[next ↦ newNext, seq ↦ delState.seq+1, label ↦ Delete(pred)]
10   if(delNode.compareAndSet(delState, delStateL)) delStateL else null
11 }
12 private def deleteWithParent(pred: Node, delNode: Node, delState: NodeState,
13     pPred: Node, parent: Node, pState: NodeState) : Boolean = {
14   assert(delState.parent == parent && pState.label.isInstanceOf[Delete])
15   val newNext = if(delState.next == null) pState.next else delState.next
16   val delStateL = labelForDelete(pred, delNode, delState, newNext)
17   if(delStateL != null){
18     helpDelete(pPred, parent, pState); completeDelete(pred, delNode, delNode); true
19   }
20   else false
21 }
22 private def completeDelete(pred: Node, delNode: Node, delStateL: NodeState) = {
23   val children = delStateL.children; val next = delStateL.next
24   if(children.isEmpty) predUpdate(pred, delNode, next)
25   else{
26     // update last child
27     val lastC = children.last; val lastCState = lastC.getState
28     if(lastCState.parent == delNode && lastCState.next != next)
29       lastC.compareAndSet(lastCState, lastCState[next ↦ next])
30     // Update predecessor
31     predUpdate(pred, delNode, children.head)
32     // Update children, changing parent from delNode to null
33     for(c <− children){
34       val cState = c.getState
35       if(cState.parent == delNode) c.compareAndSet(cState, cState[parent ↦ null])
36 } } }
```

Figure 7: The delete, labelForDelete, deleteWithParent and completeDelete functions.

show that it satisfies the following property (proved as Lemma 25). The claim includes a premiss concerning the heap in which delNode might be; recall that we currently assume that the heap of the deletion is not concurrently the giving heap in a union, so delNode can be in no heap other than the heap of the deletion; we relax this assumption in Section 11, where this stronger form is useful.

Claim 9 *Suppose delNode is marked for deletion and delNode.parentless. Consider a call to findPred(delNode) such that there is no giving union on the heap between the marking and findPred returning. Then findPred returns either:*

1. *a pair (p, pState), where pState was a state of p at some point during the call to findPred, such that pState.next = delNode and pState.parentless; in this case, p was not decoupled at the start of the call to findPred; or*

2. *the pair (delNode, null); in this case, the deletion of delNode has been completed.*

In case 1, a call to completePredUpdate (line 11 of Figure 8) attempts to decouple

```
1  private def predUpdate(pred: Node, delNode: Node, newNext: Node) = {
2    val predState = pred.maybeClearParent()
3    if(predState.next == delNode && predState.parentless &&
4         completePredUpdate(pred, predState, delNode, newNext)){ } // done
5    else{
6      while(!delNode.deleted){
7        val (p, pState) = findPred(delNode)
8        if(p == delNode){ assert(delNode.deleted); done = true }
9        else{
10         assert(p != null && pState.next == delNode && pState.parentless)
11         if(!delNode.deleted) completePredUpdate(p, pState, delNode, newNext)
12 } } } }
13 private def completePredUpdate(
14     pred: Node, predState: NodeState, delNode: Node, newNext: Node) : Boolean = {
15   assert(predState.next == delNode && predState.parentless)
16   predState.label match{
17     case null =>
18       val newPredState = predState[next ↦ newNext]
19       if(pred.compareAndSet(predState, newPredState)){ delNode.deleted = true; true }
20       else if(delNode.deleted) true
21       else{  // pred's state changed
22         val newPredState = pred.maybeClearParent()
23         if(newPredState.next == delNode && newPredState.parentless)
24           completePredUpdate(pred, newPredState, delNode, newNext) // retry
25         else false
26       }
27     case _ => help(pred, predState); false
28 } }
```

Figure 8: The predUpdate and completePredUpdate functions.

delNode from its expected predecessor p. In case 2 (line 8), the deletion has been completed by another thread.

The function completePredUpdate attempts to update pred's state from predState so that its next reference points to newNext. If this fails, and the deletion hasn't been completed by another thread, but pred's new state is such that the update is still possible, it recursively retries. It returns a boolean indicating whether the deletion has been completed.

Both predUpdate and completePredUpdate obtain pred's state via the function maybeClearParent (Figure 9), which has the side effect of clearing the parent field if the parent node has been deleted.

We now sketch an enhancement to predUpdate. The search encapsulated by findPred is quite expensive, because it has to be resilient against other threads changing the root list. We can replace it by a less expensive but unreliable search (using a traversal similar in style to the one in Section 6), that may sometimes fail to find delNode even though it has not been decoupled from the root list. In this case, we need to repeat the search for pred. However, the unreliable version could fail to find pred repeatedly, which could represent a failure of lock-freedom. To avoid this, if the unreliable search fails and a subsequent search is required, we then use the findPred search. Our informal experiments suggest that the unreliable search is about three times faster than findPred, and that the subsequent search is very rarely necessary, making this enhancement worthwhile.

4.1 Correctness

We now argue the correctness of the deletion: assuming the stated precondition, the delete function either deletes the node and returns true, or leaves the heap unchanged and returns false; deleteWithParent is similar, except it maybe helps comeplte the deletion of the parent. We assume here that Claim 9 holds; we discharge this assumption in Section 5.

Lemma 10 *The implementation of each of delete and deleteWithParent works correctly when it operates without interference from other threads.*

Proof: (Sketch.) The updates performed by a single thread, without interference, correctly update the states of the relevant nodes. □

Note also that the order in which the parent fields of the children are cleared is unimportant.

We now consider the effect of other threads helping with the deletion.

Lemma 11 *If a thread calls completeDelete after the delNode-labelling step has happened, then the behaviour is correct, either contributing to the deletion, or not interfering with it.*

Proof: Careful examination of the code shows that such helping threads contribute correctly to the deletion: each update on a node is executed only from an appropriate state, so if several threads attempt such an update, only the first will succeed. In particular, the successor and children of delNode do not change after delNode is marked, and so all calls to predUpdate receive the same value for the parameter newNext. □

Any helping thread will help via the helpDelete function in Figure 9 (perhaps via the help function from Figure 5). Note that it is fine for a thread to call helpDelete after the operation has proceeded further, or even completed: in this case, some of the thread's actions turn into no-ops.

We now consider interference from other operations (ignoring liveness considerations, for the moment).

Note that the maybeClearParent function (Figure 9) follows this rule, specifically the second case.

Lemma 12 *Other operations do not interfere with the correctness of deletion.*

Proof: If any thread updates delNode before it is marked, then the deletion fails, and the heap is left unchanged.

After delNode is marked, by Rule 3, no other operation may update it other than to help with the deletion.

Let c be a child of delNode. By Rule 1, the only updates allowed next on c either (a) help with the deletion (as noted above, the order in which the children have their parent fields cleared does not matter); or (b) mark c for deletion, which does not interfere with the deletion of delNode.

An update of delNode's predecessor —either the original predecessor, or one found subsequently— might cause the pred update CAS to fail, in which case completePredUpdate has no effect. The new predecessor might need to be found in this case. □

```
1  private def maybeClearParent() : NodeState = {
2    val oldState = state.get; val p = oldState.parent
3    if(p != null && p.deleted){
4      val newState = oldState[parent ↦ null]
5      if(compareAndSet(oldState, newState)) newState
6      else maybeClearParent() // retry
7    }
8    else oldState
9  }
10 private def helpDelete(pred: Node, helpNode: Node, helpState: NodeState) = {
11   if(!helpNode.deleted){
12     val parent = helpState.parent
13     if(parent != null){
14       val pState = parent.getState; val Delete(pPred) = pState.label
15       helpDelete(pPred, parent, pState)
16       completeDelete(pred, delNode, delNode.maybeClearParent, false)
17     }
18     else completeDelete(pred, helpNode, helpState, false)
19 } }
```

Figure 9: The maybeClearParent function (within the Node class), and the helpDelete function.

Lemma 13 *The delete operation respects Invariants 1–5, and Rules 1–3 and later rules concerning nodes with labels.*

Proof: Invariant 1 is trivially maintained, since parent references are only changed to null.

Invariant 2 is easily seen to be maintained: no node has its children or degree field changed; and the linked list of delNode's children, and the parent fields of the children are changed only after delNode is marked.

Invariant 3 is maintained since no parent reference is set to a non-null value.

The marking of delNode satisfies Invariant 4, by design, since delNode is either an unlabelled root or has a parent that is marked for deletion. Hence the marking of this node from its initial state satisfies Invariant 4. All other steps trivially satisfy this invariant.

Invariant 5 is trivially satisfied by all steps.

The marking of delNode, specifically when it is a non-root, satisfies Rule 1. The completePredUpdate function is called only if pred has a null parent, so the update on pred satisfies Rule 1. The updates on the children clearly satisfy Rule 1. Note that the maybeClearParent function (Figure 9) follows this rule, specifically the second case.

Rule 2 and later rules concerning nodes with labels are trivially satisfied by all steps, since no node with such a label is updated.

The marking of delNode trivially satisfies Rule 3, since delNode is unlabelled. The completePredUpdate function checks that pred is unlabelled, and so satisfies this rule. The setting of delNode's deleted field satisfies the second case of the rule. The updates on the children satisfy the first case of the rule (or vacuously). □

The following invariant concerns next references, including circumstances under which next references from multiple different nodes can point to the same node. Once a node has been decoupled, its next reference will not change, but become fairly irrelevant. We concentrate below on next references from nodes

that have not been decoupled. We call these "non-decoupled next references". In particular, the invariant shows that once a node is decoupled, there are no "dangling" references to it.

Invariant 6 *1. Suppose there are two non-decoupled next references from nodes n_1 and n_2 to the same node n. Then n_1 and n_2 are the parent and last child of each other, with the parent marked for deletion. Hence there cannot be a third non-decoupled next reference to n: the last child has degree 0 so itself has no children.*

2. *Suppose there is a non-decoupled next reference from node n to p's first child, p.children.head. Then either:*
   (a) *p is marked for deletion and has been decoupled; or*
   (b) *p if the a node of a merge and n is the b node, and the b update but not the a update has taken place (so p has an appropriate MergeParent label).*
3. *There is no non-decoupled next reference to a decoupled node.*

Lemma 14 *Invariant 6 is satisfied by merge and delete operations.*

Proof: Consider, first, merge operations.

- The b update establishes the condition in item 2b. Previously there was no non-decoupled next reference to p.children.head (by item 2 applied inductively), so this maintains item 1.

  The node previously referenced by b.next now has no non-decoupled next reference to it (by item 1, applied inductively). It is easy to check that no new reference is made to it before the pred update.
- The pred update removes the non-decoupled next reference to b. This reference was previously unique (by item 1 applied inductively), so subsequently b has no non-decoupled next reference to it. It is easy to check that no new non-decoupled next reference is made to it before the merge is complete.

  pred.next is updated to point to the node previously referenced by b.next; this non-decoupled next reference is unique, by the previous item.
- The a update changes p's first child to be the b node; there is no non-decoupled next reference to this node, by the previous item. Hence this maintains item 2, vacuously.

Now consider a delete operation on node d. Let $n = d$.next. Suppose, first, that d has at least one child.

- The update of d's last child creates a second non-decoupled next reference to n, but this is consistent with item 1. Note that either d has a null parent, or d is not its parent's last child (since the last child has no children); so there is not a third non-decoupled next reference pointing to d.next (by item 1 applied inductively). Likewise, if d's last child is marked for deletion, its next reference is again updated to n.
- The pred update is applied only to an unlabelled root node *pred* such that *pred*.next $= d$. By item 1, there is no other non-decoupled next reference to d. Hence this decoupling of d establishes item 3.

The pred update creates a next reference to d's first child c, consistent with item 2a; this is unique, by item 2, applied inductively.

Further, by item 3 applied inductively, no subsequent next reference to d is created. Hence a single pred update corresponding to d is performed, so a single such next reference to c is created.

This decoupling of d maintains item 1 for n.

- The clearing of the parent references of the children maintains both parts. In particular, this happens after d is decoupled, so item 1 is maintained.

If d has no children, then the pred update creates a second next reference to n; but this action decouples d, so this is consistent with item 1. As above, the pred update is applied to a unique node.

It is easy to see that once item 3 is established for a node d, no new non-decoupled next reference to d is created, so this item remains true. □

The following proposition brins together the previous results.

Proposition 15 *Assuming the stated preconditions, the delete function either deletes the node and returns true, or leaves the heap unchanged and returns false. The deleteWithParent satisfies a similar property, except may additionally help with the deletion of the parent.*

We now prove a partial liveness result for delete, deleteWithParent and completeDelete. This lemma assumes limited interference from other threads. In Section 14, we will build on this to prove lock-freedom. The lemma considers only steps performed outside findPred and the helping of another operation. In Section 5.4 we will show that, under similar assumptions, calls to findPred take a finite number of steps. We will then close the loop, and show that calls to these functions take a finite number of steps in total.

Lemma 16 *Suppose a call to either delete, deleteWithParent or completeDelete occurs such that, from some time on, no other thread makes an update to any node. Then the execution performs a finite number of steps outside the call to findPred within predUpdate (line 7 of Figure 8), and the attempt to help either parent within deleteWithParent (line 18 of Figure 7), or pred within completePredUpdate (line 27 of Figure 8).*

Proof: Consider part of an execution such that no other thread makes an update to any node.

First, note that CAS operation in maybeClearParent will succeed (under the given assumptions), so each call to this function will perform a finite number of steps.

Now consider completePredUpdate. A call can fail at line 23 only if pred's state changes, which contradicts the assumptions of this lemma. For the same reason, the call cannot recurse. Therefore each call performs a finite number of steps outside the call to help.

If pred is labelled, then the call to completePredUpdate helps with that operation, and fails. If the label corresponds to a merge, then, by Lemma 8, this can happen only a finite number of times for each pred before the operation is completed or backtracked. If the label is a Delete, then the first call to help (assuming it returns) completes the deletion, uncoupling the node.

Now consider a call to predUpdate. The **while** loop repeats only if completePredUpdate fails because the predecessor was labelled. By the above, this can happen only a finite number of times for each label corresponding to a merge. Also by the above, in the case of a Delete label, that node becomes decoupled; hence a subsequent call of findPred will not find the same node (by Claim 9), and so completePredUpdate will not be called again on the same node. Further, note that, by the assumptions, no new node will receive a label. This gives a finite number of iterations in total, and so this call satisfies the result of the lemma.

Finally note that delete and completeDelete contain no recursion, and the only loop is the iteration over children in completeDelete, which is finite (by Invariant 2), so these perform a finite number of steps. The same argument holds for deleteWithParent outside the attempt to help parent. □

5 Finding the predecessor

We now describe how the root list can be traversed in order to find the predecessor of a node that is being deleted. We use a very similar traversal method in Sections 8 and 9 to find the node with the smallest key. Recall, from the Introduction, that straightforward approaches risk missing relevant nodes if the root list is rearranged concurrently.

As we traverse the root list, when we first encounter a new root curr, we record its sequence number currSeq. If we subsequently find that curr's sequence number has changed —indicating that in the mean time curr has been merged below another node, or had a Delete label added— then we restart the traversal. Conversely, if the sequence number has not changed, the node has been a root throughout the intervening period.

In addition, while traversing, we skip over any node dn with a Delete label: otherwise, once the deletion was complete, other threads could move dn.next around the heap; but the traversing thread would not notice this, since the state of dn itself would not change.

In Section 5.1 we present a subsidiary function advance that advances one step along the root list; we prove relevant properties. Then in Section 5.2, we present the findPred function, used by delete to find the predecessor of the node being deleted. We prove a correctness property in Section 5.3, in particular that if the traversal does not find the node being deleted, then another node has decoupled it. We prove a liveness property in Section 5.4.

We need to clarify what we mean by a root node. This is clear when there are no partially completed updates, but less clear otherwise. Informally, we consider a merge to take effect at the b update: after this update, we define b to no longer be a root. Similarly, we consider a delete to take effect when the node being deleted is decoupled: after this update, we define the node being deleted to no longer be a root. The following definition captures this formally.

Definition 17 *We define a function* $R : \mathit{Node} \to \mathit{List}(\mathit{Node})$ *as follows*[9]*, such that* $R(n)$ *gives the list of root nodes starting from* n*.*

- $R(\mathit{null}) = \mathit{List}()$.

[9]We use Scala notation for lists: $\mathit{List}(a, b, \ldots, z)$ represents a list containing $a, b, \ldots, z$; "::" represents cons; "++" represents list concatenation.

- *If n has a MergeNext(a, b, bState) label or a MergeParent($n, \_, \_, b$, bState) label (so, in each case, n.next $= b$, by Invariant 5), and b.parent $= a$ (so the b update has occurred), then $R(n) = n :: R(b\text{State.next})$.*
- *Otherwise, $R(n) = n :: R(n.\text{next})$.*

The root list *of a heap is defined to be $R(\text{head})$. We say that r is a* root *if it is a member of the root list.*

Note that the root list might contain partially deleted nodes, which have been given a Delete label but not yet decoupled.

Lemma 18 *No root node is decoupled. Hence no root node is deleted.*

Proof: This is a straightforward induction, using item 3 of Invariant 6. □

Invariant 7 *The root list is finite: in particular, it is acyclic.*

Proof: The root list initially contains a single node. Deleting a node of degree d —in particular the pred update— replaces the deleted node by its d children (cf. Invariant 2). A merge —in particular the b update— removes a single node from the root list. We will see in Section 6 that an insertion might add a single new node at the end of the root list; and we will see in Section 7 that the union operation concatenates the argument heap's root list onto the current heap's root list. No other step of any operation changes the root list. □

The following lemma will be useful below. Recall that we currently do not allow giving unions (i.e. unions where this is the giving heap) concurrent to other operations on the same heap; we relax this in Section 11.

Lemma 19 *Suppose node n, at some point in time, was a root node of heap h, had sequence number seq, and did not have a Delete label. If subsequently it still has sequence number seq, and there has been no intervening giving union on h, then it is still a root node of h, and still does not have a Delete label.*

Proof: When a node has a Delete label added, its sequence number is increased. Hence, if its sequence number is unchanged, it cannot have had a Delete label added.

The only point at which a node without a Delete label changes from being a root to being a non-root is at the b update of a merge. At this point, the sequence number is incremented.

By assumption, the node has not been transferred to another heap by a giving union, so must still be in h. □

5.1 Advance

We use a subsidiary function advance (Figure 10) to help advance along the root list. It takes the current node curr and its expected sequence number currSeq; curr is assumed not to be marked for deletion unless its sequence number has changed (line 10). It normally returns a triple (next, nextSeq, skipNodes), where next is a subsequent root node (or null if the end of the root list has been reached), nextSeq is next's sequence number (or, arbitrarily, -1 if next is null), and skipNodes is a list of root nodes between curr and next that have either been

```
1  private def advance(curr: Node, currSeq: Int): (Node, Int, List[Node]) = {
2    var result : (Node, Int, List[Node]) = null
3    while(true){
4      val currState = curr.getState
5      if(currState.seq != currSeq) return null
6      else{
7        currState.label match{
8          case MergeParent('curr', _, _, b, bState) => result = skipMerge(curr, b, bState)
9          case MergeNext(a, b, bState) => result = skipMerge(a, b, bState)
10         case Delete(_) => assert(false)
11         case _ => result = skipDeleted(currState.next)
12       } // end of match
13       if(curr.getState == currState) return result
14 } } }
15 private def skipDeleted(n: Node): (Node, Int, List[Node]) = {
16   if(n == null) (n, -1, List())
17   else{
18     val nState = n.getState
19     if(nState.label.isInstanceOf[Delete]){
20       val (next, nextSeq, dels) = skipDeleted(nState.next); (next, nextSeq, n::dels)
21     }
22     else (n, nState.seq, List())
23 } }
24 private def skipMerge(a: Node, b: Node, bState: NodeState): (Node, Int, List[Node]) = {
25   val (next, nextSeq, dels) = skipDeleted(bState.next); val myBState = b.getState
26   if(myBState == bState) (next, nextSeq, b::dels)
27   else if(myBState.parent == a) (next, nextSeq, dels)
28   else if(myBState.label.isInstanceOf[Delete]){
29     val (next1, nextSeq1, dels1) = skipDeleted(myBState.next); (next1, nextSeq1, b::dels1)
30   }
31   else (b, myBState.seq, List())
32 }
```

Figure 10: The advance, skipDeleted and skipMerge functions.

marked for deletion or that are in the process of being merged below another node. We will show that, assuming curr was a root node, next was (at some point during the call of advance) also a root node, not marked for deletion, and had sequence number nextSeq, and that skipNodes held the intermediate root nodes. By Lemma 19, next remains a root node for as long as its sequence number is nextSeq. advance returns null (line 5) if the current node's sequence number has changed, to indicate a failure; in this case, the traversal restarts.

The subsidiary function skipDeleted(n) advances along the root list, skipping over nodes that have been marked for deletion. It returns a triple (next, nextSeq, dels) where next is the first root not marked for deletion (or null if there is no such), nextSeq is its sequence number (or -1 if next is null), and dels is the list of nodes marked for deletion between n and next.

In most cases, advance returns the result of calling skipDeleted on curr.next. However, if curr's state changes after it is initially read (at line 4), then the call to advance restarts: this is necessary to be sure that the next node is still a root node at the point that its sequence number is read. Note, though, that the traversal does not need to restart in this case (unless the sequence number is subsequently found to have also changed).

The cases where the current node has a MergeNext label, or a MergeParent label corresponding to the case curr = a = pred, are handled by the function

skipMerge. If the b update has happened (line 27) then we skip over the next node (b), following the definition of the root list. Otherwise, if the state of b is unchanged (line 26), then, for pragmatic reasons, we again skip over b: we avoid alighting on b since it will probably be updated soon, which could disrupt the traversal; however, we return b within the third component of the result. Finally, if the state of b has changed (lines 28–31), we just advance in the normal way (the result here is equivalent to skipDeleted(b), but avoids repeating work).

Lemma 20 *Suppose skipDeleted is called on a node n ≠ null which is a root node of the current heap throughout the call, and suppose the call returns a triple (next, nextSeq, dels). Then at some time t during the call:*

1. *If next ≠ null then next was a root node, had sequence number nextSeq, and was not marked for deletion.*
2. *The nodes returned were consecutive roots: R(n) = dels ++ R(next). Further, this remains true for as long as n remains a root.*

Proof: We perform an induction on the number of recursive calls made.

- Suppose n is not marked for deletion, so skipDeleted returns from line 22. Let t be the time of the read of n.getState. The result follows immediately.
- Suppose n is marked for deletion and nState.next ≠ null. Consider the recursive call on nState.next. The value of nState.next doesn't change (Rule 3), and so, if nState.next ≠ null, that node remains a root node for at least as long as n, so at least throughout the recursive call. Hence, by induction, $R(\mathsf{nState.next}) = \mathsf{dels} +\!\!+ R(\mathsf{next})$ at some time t. Hence $R(\mathsf{n}) = \mathsf{n} :: \mathsf{dels} +\!\!+ R(\mathsf{next})$, as required.
- Suppose n is marked for deletion and nState.next = null. Then this call returns (null, -1, $List(n)$), and the result holds.

□

The following lemma captures the main properties of advance.

Lemma 21 *Suppose advance is called on a node curr that at some point in the past was a root node of heap h, had sequence number currSeq, and did not have a Delete label; and suppose there is no giving union on h concurrent with the call. Suppose advance returns a triple (next, nextSeq, skipNodes). Then at some time t during the call:*

1. *If next ≠ null then next was a root node of h, had sequence number nextSeq, and was not marked for deletion.*
2. *The nodes returned were consecutive roots:*

$$R(curr) = List(curr) +\!\!+ skipNodes +\!\!+ R(next).$$

Proof: Assume the conditions of the lemma. Since advance returns a non-null result, the check "curr.getState == currState" must have returned true; let t_c be the time of this check. Then, since currState.seq = currSeq, curr was a root at t_c and all earlier times within this call to advance, by Lemma 19. Also by Lemma 19, curr cannot have a Delete label, so line 10 does not raise an exception. We perform a case analysis.

- In the default case, if currState.next = null, advance returns (null, −1, List()) (via skipDeleted), and the result follows easily. Otherwise, currState.next was the root following curr throughout the call to skipDeleted. The result follows from Lemma 20.
- Consider the case of a MergeNext label, or a MergeParent label corresponding to the case a = pred; note that b = curr.next.
  - Suppose the state of b had not changed (line 26), so b is still a root. Then bState.next was the next root after b throughout the call to skipDeleted. The result then follows from Lemma 20.
  - Consider the case that the b update had happened at the point of the b.getState (line 27). Then bState.next was a root throughout the call to skipDeleted: the next root after curr after the b update, or the next root after b before it. The result again follows from Lemma 20, in particular at the point of the read of b.getState.
  - Otherwise (lines 28–31), the proof is as in the default case: the result returned is equivalent to skipDeleted(b).

□

The following lemma will prove useful later.

Lemma 22 *Suppose no changes are made to the root list during a call to advance(curr, currSeq), and the call returns a triple* $(next, nextSeq, skipNodes)$. *Then either:*

- *the nodes of* $List(curr) +\!\!+ skipNodes +\!\!+ R(next)$ *are linked together via their next pointers; or*
- *the nodes of* $skipNodes +\!\!+ R(next)$ *are linked together via their next pointers, curr has a MergeNext label, or a MergeParent label, and the b update has happened.*

Proof: The proof is a straightforward check of the different branches. It uses the additional results (assuming no changes to the root nodes):

- if a call to skipDeleted(n) returns a triple $(next, nextSeq, dels)$ then the nodes of $dels +\!\!+ R(next)$ are linked together via their next pointers;
- when skipDeleted is called, b = curr.next (by Invariant 5).

□

5.2 findPred

We now describe the function findPred (Figure 11), used by delete to find the predecessor of the node delNode being deleted if the latter is still in the heap. The traversal keeps track of the current node curr and its sequence number currSeq. It uses advance to advance along the root list. If advance returns null, indicating that curr's sequence number has changed, the traversal restarts. Otherwise, a search is made through skipNodes for delNode and the node pred that was before it during the call to advance (line 8). If it finds delNode, and pred is still the previous root, pred and its state are returned (line 19).

If pred is not the previous node, it is possible that pred had a MergeNext or MergeParent node, and the b update had happened, so advance skipped over the b

```
1  private def findPred(delNode: Node) : (Node, NodeState) = {
2    var curr = head; var currSeq = curr.getSeq
3    def restart() = { curr = head; currSeq = curr.getSeq }
4    while(curr != null) advance(curr, currSeq) match{
5      case null => restart()
6      case (next, nextSeq, skipNodes) =>
7        var pred = curr; var sn = skipNodes
8        while(sn.nonEmpty && sn.head != delNode){ pred = sn.head; sn = sn.tail }
9        if(sn.nonEmpty){ // sn.head == delNode
10         var pState = pred.maybeClearParent()
11         if(pState.next != delNode){
12           if(pred == curr && pState.label != null){
13             help(pred, pState); pState = pred.maybeClearParent()
14             if(pState.next == delNode && pState.parentless) return(pred, pState)
15             else restart()
16           }
17           else restart()
18         }
19         else if(pState.parentless) return(pred, pState)
20         else{ // need to help pState.parent
21           val parent = pState.parent; val parentState = parent.getState
22           val pPred = parentState.label.asInstanceOf[Delete].pred
23           helpDelete(pPred, parent, parentState); pState = pred.maybeClearParent()
24           if(pState.next == delNode && pState.parentless) return(pred, pState)
25           else{ assert(pState.parent != parent); restart() }
26         }
27       }
28       else{ curr = next; currSeq = nextSeq }
29   } // end of while loop
30   delNode.deleted = true; (delNode, null)
31 }
```

Figure 11: The findPred function.

node. If the pred update has not yet happened (cf. Lemma 22), so pred still points to b, it is necessary to help with the merge, to ensure lock-freedom (line 13). If pred is now the predecessor of delNode, the function returns; otherwise the traversal restarts.

Another complication arises if pred has a non-null parent parent. The most likely cause was that parent was decoupled from the root list before the traversal reached pred, but its deleted field has not yet been set. However, it's also possible that after the traversal reached pred, it was merged as the last child of parent, which was previously its predecessor, then parent was marked for deletion, and pred.next updated to point to delNode again. Thus we cannot assume that parent has been decoupled. We help complete the deletion of parent, in order to ensure lock-freedom. If pred is now the predecessor of delNode, the function returns (line 24); otherwise it restarts the traversal.

If skipNodes does not contain delNode, the traversal advances (line 28). If the traversal reaches the end of the root list without finding delNode, it must have been detached by another thread: this thread sets the node's deleted field, and returns (delNode, null) to indicate that it has been detatched (line 30).

5.3 Correctness

If r is a root node, we write $rootsUpTo(r)$ for the subsequence of the root list up to and including r, but excluding the dummy header.

Definition 23 *If the root list is of the form* $List(\mathit{head}) +\!\!+ xs +\!\!+ List(r) +\!\!+ ys$, *then* $rootsUpTo(r) = xs +\!\!+ List(r)$. *We define* $rootsUpTo(\mathit{head}) = List()$. *For convenience, we define* $rootsUpTo(\mathit{null})$ *to be all the root nodes excluding* head.

The main invariant of the loop in findPred is that delNode does not appear in $rootsUpTo(\mathsf{curr})$. The following lemma gives sufficient conditions for this property to be preserved by steps of other threads.

Lemma 24 *Suppose node* d *is marked for deletion,* $d.\mathit{parent} = \mathit{null}$, *and* $d \notin rootsUpTo(r)$. *Consider a step of an operation that does not change the sequence number of* r. *Then subsequently still* $d \notin rootsUpTo(r)$.

Proof: We perform a case analysis over the actions of threads that affect the root list but do not change the sequence number of r. Note that, by Lemma 19, r remains a root, so $rootsUpTo(r)$ is still well defined.

- The b update of a merge (necessarily not on r) can only remove an *unlabelled* node from the root list, so maintains the property.
- The pred update of a deletion removes the node being deleted from the root list, and adds its children. Since $d.\mathsf{parent} = \mathsf{null}$, d is not among the children.
- An insert or receiving union adds new nodes at the *end* of the root list, so not within $rootsUpTo(r)$.

□

We can now verify findPred (presented earlier as Claim 9).

Lemma 25 *Suppose* delNode *is marked for deletion and* delNode.parentless. *Consider a call to* findPred(delNode) *such that there is no giving union on the heap between the marking and* findPred *returning. Then* findPred *returns either:*

1. *a pair* (pred, pState), *where* pState *was a state of* p *at some point during the call to* findPred, *such that* pState.next = delNode *and* pState.parent = null; *in this case,* pred *was not decoupled at the start of the call to* findPred; *or*
2. *the pair* (delNode, null); *in this case, the deletion of* delNode *has been completed.*

Proof: Note by the premiss about giving unions, delNode is in the heap unless it has been decoupled.

We first show that the following is invariant: if $\mathsf{curr} \neq \mathsf{null}$ and $\mathsf{curr.getSeq} = \mathsf{currSeq}$, then curr is a root node and has not been marked for deletion. This is true initially and is re-established whenever the traversal restarts. By Lemma 19, no action by another thread falsifies this property. Consider a call to advance that returns a result (next, nextSeq, skipNode). By part 1 of Lemma 21, at some point during that call of advance, next was a root node, had sequence number nextSeq, and did not have a Delete label; and, again, by Lemma 19, no action by another thread falsifies this property unless it changes the sequence number. Hence line 28 maintains this invariant.

Now consider the returns at lines 19 and 24. These clearly satisfy that pState.next = delNode and pState.parent = null. To satisfy the first disjunct, we show that pred was not decoupled. The value of pred equals either curr or a member of skipNodes. We have already shown that the former was a root when it was encountered; and by Lemma 21, each of the latter was a root at some point during the last call to advance. Hence in each case, pred was not decoupled before the start of the call.

We now show that the return at line 30 satisfies the second disjunct, by showing that in this case delNode has been decoupled from the heap. We show that the following is invariant: if curr = null or curr.getSeq = currSeq, then delNode $\notin$ *rootsUpTo*(curr). Again, this is true initially and is re-established whenever the traversal restarts. By Lemma 24, any action of another thread that removes delNode from *rootsUpTo*(curr) also changes curr's sequence number. Consider a call to advance that returns a result (next, nextSeq, skipNode). From Lemma 21, at some point during the call to advance:

$$rootsUpTo(\mathsf{next}) \quad = \quad rootsUpTo(\mathsf{curr}) +\!\!+ \mathsf{skipNodes} +\!\!+ List(\mathsf{next}).$$

The code checks that delNode is not a member of skipNodes. Necessarily, delNode $\neq$ next, since delNode is marked for deletion, but next isn't. Hence the update of curr at line 28 maintains this invariant.

Hence, we can deduce that when the **while** loop terminates with curr = null, delNode is not a root node, and hence has been decoupled from the heap. The update to delNode.deleted completes the deletion. □

5.4 Liveness

We now present a liveness result. We again assume limited interference from other threads.

Lemma 26 *Consider an invocation of findPred, during which other threads make a finite number of updates to nodes. Then the invocation performs a finite number of steps, except, perhaps, within the call to helpDelete (line 23).*

Proof: Suppose, from some point onwards during the invocation, no other thread makes any change to any node. Then each call to advance terminates on its first iteration (since curr's state does not change between its two reads). Further, since the sequence numbers do not change, and each call receives the unchanged sequence number, each call returns a non-null result.

Now consider the traversal. Since advance always succeeds, and the root list is finite (Invariant 7), the main loop either reaches the end of the root list and so returns, or finds delNode and either returns or restarts.

We now show that if the *whole* of a traversal is after other threads stop changing nodes, and it finds a node delNode, then findPred returns.

- If pred is the root before delNode, then the function returns, as required.
- If pred.next $\neq$ delNode then by Lemma 22, it must be the case that pred = curr, curr has a MergeNext or MergeParent label, and the b update has happened. In this case, the function helps with the merge (line 13), and then either returns or restarts. By Lemma 8, this thread has to help with the merge a finite number of times, and each instance terminates in

a finite number of steps. The number of nodes that have to be helped in this way is bounded by the number of root nodes.

- If pred has a non-null parent, then that parent has been marked for deletion. findPred helps with that deletion, as allowed in the statement of this lemma. Assuming this returns, the deletion of the parent has been completed, except, perhaps, the clearing of its children's parent fields. By the assumption about lack of interference, the call to pred.maybeClearParent clears pred's parent field, and then the function returns at line 24.

□

The following result follows directly from Lemmas 16 and 26. We build on it to prove lock-freedom in Section 14; in particular, we show that any chain of helping is finite.

Proposition 27 *Consider an invocation of either delete, deleteWithParent or completeDelete, during which other threads make a finite number of updates to nodes. Then the invocation returns after a finite number of steps other, perhaps, than steps helping other operations.*

We now sketch a small enhancement to findPred. It is clear that the traversal can reach a child of delNode, or delNode.next, only after delNode has been decoupled. Hence we can use either delNode's first child or delNode.next as a sentinel in the guard of the while loop: if the sentinel is found, we can again complete the deletion (at line 30).

6 Insertion

Inserting a new key k can be done by adding a new node containing k either (a) after the final root node, or (b) as the child of a root of degree 0 whose key is no larger than k. The former approach is the more obvious. We include the latter approach for two reasons. Firstly, it avoids the last root node becoming a bottleneck. Secondly, it shortens the root list, and in particular means that if another thread is traversing the root list, it isn't racing against more and more root nodes being added.

The insert function is in Figure 12. It traverses the root list, keeping track of the current node in curr. It is possible that this traversal is disrupted by other threads rearranging the heap, but this does not matter in this case.

It is possible that the traversal reaches a node with a non-null parent (line 12): another thread might have merged curr below another node, or partially deleted curr's parent; in this case, the parent reference is followed.

If the end of the root list is reached, and the last node is unlabelled, the operation tries to append the new node after it (line 17). If the appending CAS is unsuccessful, this thread spins, using a binary back-off, to avoid creating memory conflicts. If the last node has a label, this thread helps with the corresponding operation.

If the traversal reaches an unlabelled root node of degree 0 whose key is no larger than k (line 19), the operation tries to insert the new node below it.

Otherwise, the next reference is normally followed to continue the traverse. However, if the current node has a MergeNext label (or a MergeParent label where the following node is the one being merged) then it is often advantageous to skip

```
def insert(key: Int) = {
  val myNode = new Node(key); var curr = head; var currState : NodeState = null
  // Try to skip over the next node if it is being merged.
  def skip(a: Node, b: Node, bState: NodeState) = {
    val myBState = b.getState; val next = bState.next
    if (next != null)
      if (myBState == bState || myBState.parent == a) curr = next else curr = b
    else if (myBState.parent == a) help(curr, currState) else curr = b
  }
  while(true){
    currState = curr.maybeClearParent()
    if (currState.parent != null) curr = currState.parent
    else if (currState.unlabelled){
      if (currState.next == null){ // at last node
        val newCurrState = currState[next ↦ myNode]
        if (curr.getState == currState)
          if (curr.compareAndSet(currState, newCurrState)) return else Backoff()
      }
      else if (currState.degree == 0 && curr.key <= key){ // insert below curr
        assert(curr != head); myNode.state = myNode.state[parent ↦ curr]
        val newCurrState = currState[degree ↦ 1, children ↦ List(myNode)]
        if (curr.compareAndSet(currState, newCurrState)) return
        else myNode.state = myNode.state[parent ↦ null] // might retry on next iteration
      }
      else curr = currState.next
    } // end of if(currState.unlabelled)
    else currState.label match{
      case MergeNext(a, b, bState) => skip(a, b, bState)
      case MergeParent(pred, _, _, b, bState) =>
        if (curr == pred) skip(curr, b, bState)
        else if (currState.next != null) curr = currState.next else help(curr, currState)
      case delLabel @ Delete(pred) =>
        if (currState.next != null) curr = currState.next
        else{ helpDelete(pred, curr, currState, delLabel); curr = head }
} } }
```

Figure 12: The insert function.

```
def clear = state.set(
  new NodeState(parent = null, degree = 0, children = List(),
                next = null, seq = 0, label = null))
def setParentInitial(p: Node) = state.set(
  new NodeState(parent = p, degree = 0, children = List(),
                next = null, seq = 0, label = null))
```

Figure 13: The clear and setParentInitial functions of Node.

over the next node, which is about to be merged below another (similarly to as in advance, Section 5): indeed, if the b update has occurred, this is necessary in some circumstances to avoid getting stuck in a loop, and so to ensure the implementation is lock-free. This is implemented using the function skip: the b-node of the merge is skipped over, unless the merge has been interfered with and so can't complete.

A small enhancement is that if the last root is involved in another operation, it can be more efficient *not* to help that operation, but instead to restart the traversal, in the hope the operation completes anyway: our approach is to restart

the first two times, and subsequently to restart with some probability; informal experiments suggest a probability of $\frac{7}{8}$ works well.

6.1 Correctness

We seek to prove that the new node is correctly inserted, either after the last root, or as the child of a root. We need some additional results.

Definition 28 *We say that a node n is a* strict descendant *if, starting at n, following parent references one or more times reaches a root node; i.e., using obvious notation, $n(.parent)^k$ is a root for some $k \geq 1$.*

We say that a node is new *if it has been created by the insert function but not yet added to the heap.*

Note that by Invariant 4, each strict descendant has a null label.

Invariant 8 *Nodes are partitioned into root nodes, strict descendant nodes, decoupled nodes, and new nodes.*

Proof: The dummy header node is initially a root node. Other nodes are initially created as new nodes, and become root nodes when they are added to the end of the root list, or strict descendant nodes if they are added as the child of another node. The pred update of a deletion of delNode changes delNode from a root to decoupled, and changes each of delNode's children from strict descendants to roots. The b update of a merge changes b from a root to a strict descendant. No other step of any operation changes the status of any node. □

Invariant 9 *Suppose node n does not have a Delete label and is not new. Then n is a root node if and only if $n.parent$ is null or points to a decoupled node.*

Proof: When a new root node is inserted at the end of the root list, it has a null parent. If it is inserted as the child of another node, it is a non-root and has a non-null parent. If node n becomes a root as the result of the decoupling of its parent, in particular at the pred update, then its parent field points to a decoupled node. The clearing (in completeDelete) of the parent fields of the children of a fully deleted (and so decoupled) node clearly maintains the property, as does the maybeClearParent function. At the b update of a merge, the b node becomes a non-root and its parent field becomes non-null. No other step changes a node from a root to a non-root or vice versa, or changes its parent field. □

Note that the above two invariants would not hold if, within completeDelete, we performed a *single* update on the last child, to set its next field *and* clear its parent field (before the pred update).

Lemma 29 *If a node n has null next, parent and label fields, and n is not new, then it is the final root.*

Proof: Suppose n has null next, parent and label fields, and is not new. It cannot be decoupled, since such nodes have a Delete label. If cannot be a strict descendant, since such nodes have a non-null parent field. Hence it must be a root. Since its next field is null, it must be the last root, from the definition. □

Proposition 30 *If* insert *terminates, it correctly inserts* key*.*

Proof: Suppose insert inserts the new node at line 17 of Figure 12, after curr. Then by Lemma 29, curr was the last root node. Alternatively, suppose insert inserts the new node at line 22, as the child of curr. Then by Invariant 9, curr was a root, which has now become the root of a tree of degree 1. We linearize the operation at the point of the successful CAS in each case. □

Lemma 31 *insert maintains the invariants, and follows Rules 1–3.*

Proof: Invariant 1 is maintained by design, since a new node is inserted below only a node with a key that is no larger. Invariants 2–7 are clearly maintained. We have already checked Invariants 8 and 9.

It obviously follows the rules. □

6.2 Liveness

In order to prove lock-freedom, we aim to prove that insert terminates assuming a bounded amount of interference. Our approach is to identify a *rank*, from a well ordered set, that is reduced by each iteration (assuming no interference). The rank is composed of several parts, which we introduce below, together with additional results to show that the rank is indeed decreased.

If the traversal just traverses the root list, then it is clearly bounded (cf. Invariant 7). However, things are complicated by the fact that if a node d has been decoupled but not deleted, the traversal can reach d's first child, and then follow that child's parent field to d. Nevertheless, the following quantity is reduced by each such step.

Definition 32 *Consider a state of a binomial heap. We define the* next *rank of a node n, written $nextRank(n)$, as follows:*

- *For convenience, we define $nextRank(\mathit{null}) = 0$.*
- *Suppose n is a root node, and n' is the following root, or $n' = \mathit{null}$ if n is the final root; then $nextRank(n) = 2 + nextRank(n')$; so $nextRank(n)$ is twice the number of root nodes from n onwards.*
- *If n is decoupled but not deleted, then we define $nextRank(n) = 1 + nextRank(n.\mathit{next})$.*
- *Otherwise, we define $nextRank(n) = \infty$.*

Lemma 33 *Suppose node d is decoupled but not fully deleted, and has at least one child, and suppose $d.\mathit{next} \neq \mathit{null}$. Then $d.\mathit{next}$ is a root node.*

Proof: Let c be d's last child. d.next was a root node when d was decoupled. Immediately after, c.next $= d$.next, so d.next was the next root after c. Further, c's parent field points to d, because it is cleared only after d is deleted; and c's next field cannot be changed until after its parent field is cleared. Hence d.next is still a root node. □

Lemma 34 *Suppose n is a root node, and $p = n.\mathsf{parent}$ is decoupled, but not fully deleted. Then $nextRank(p) < nextRank(n)$.*

Proof: If p is decoupled but not yet fully deleted, then each of its children still has its parent field pointing to p, since these fields are cleared only after p is deleted. Further, p's children are linked together by their next references (Invariant 2): these next references are not changed until that child's parent is cleared. Further, as in the proof of Lemma 33, the last child's next reference points to $p.\mathsf{next}$ (which might be null). Thus, the root list, from n, follows the remainder of p's children to $p.\mathsf{next}$. Hence $nextRank(n) \geq 2 + nextRank(p.\mathsf{next})$, but $nextRank(p) = 1 + nextRank(p.\mathsf{next})$. □

The traversal might reach a node that is subsequently deleted. From there, it follows the next reference, to reach a node that might also have been subsequently deleted. We need to show that this cannot continue indefinitely, assuming that eventually no more nodes are deleted. We need the following definition.

Definition 35 *Consider a particular execution and a particular point in that execution. Suppose each of nodes $n_0, n_1, \ldots, n_{k-1}$ either has no children and has been decoupled, or has been fully deleted; and suppose their pred update steps were performed in the above order. Define the* deletion rank *of a node n to be*

$$\begin{array}{cc} k-i & \text{if } n = n_i, \text{ for } i = 0, \ldots, k-1, \\ 0 & \text{if } n \neq n_0, n_1, \ldots, n_{k-1}. \end{array}$$

Note that nodes that have earlier pred updates have larger deletion indices.

Lemma 36 *Suppose node d either has no children and has been decoupled, or has been fully deleted. Then $d.\mathsf{next}$ has a smaller deletion rank.*

Proof: Note that the value of $d.\mathsf{next}$ cannot change after d is marked for deletion (Rule 3). Consider the pred update step in the deletion of $d.\mathsf{next}$ (if any). This cannot use d as the predecessor, since the predecessor must be unlabelled (line 27 of Figure 8). Hence d must have been decoupled before $d.\mathsf{next}$, and so $d.\mathsf{next}$ has a smaller deletion rank than d. □

Definition 37 *We define the* depth *of a node n to be the number of parent references that can be followed to reach a root node; if n is decoupled or fully deleted, we define its depth to be 0. Note that this depth is finite, by Invariant 3.*

Lemma 38 *Consider an invocation of insert, during which other threads make a finite number of updates to nodes. Then insert terminates in a finite number of steps, outside of helping.*

Proof: Define the *rank* of the current node curr to be a triple:

(the deletion rank of curr, the depth of curr, the next rank of curr).

We order ranks lexicographically; note this is a well founded order.

Consider an iteration of the loop of insert starting from curr, operating without interference, as captured in the statement of the proposition. Note that, before the interference ends, the traversal might have reached a deleted node, from where the next step could reach an arbitrary node in the heap, or indeed another deleted node; the proof below needs to deal with all such cases. We show that each iteration except a helping iteration or the final iteration reduces the rank. We perform a case analysis over such iterations.

- If curr has a non-null parent, and its parent is not fully deleted, then its parent field is followed. Note that the use of maybeClearParent ensures that a parent pointer cannot be followed to a node that was fully deleted when currState was read; the assumption means that currState.parent cannot have been fully deleted subsequently.

  If curr is not a root node, so its parent is not decoupled, this reduces the depth by 1. If curr is a root node, so its parent is decoupled, this leaves its deletion rank unchanged at 0, leaves the depth unchanged at 0, but reduces its next rank, by Lemma 34.
- If curr is a root node, but not the last root, then the iteration advances at least one step along the root list, so reduces the next rank; in the function skipOverMerge, if myBState = bState, then it advances two steps along the root list. Note that the next node is not decoupled, so has deletion rank 0, and is a root node so has depth 0.
- Suppose curr is decoupled but not fully deleted, and has at least one child. If curr.next is not null, the iteration advances to curr.next. Then by Lemma 33, curr.next is a root node, so has deletion rank 0 and depth 0. This step reduces the next rank.
- Suppose curr either has no children and has been decoupled, or has been fully deleted. Again, if curr.next $\neq$ null, then the iteration advances to curr.next. This has a smaller deletion rank, by Lemma 36.

If the iteration attempts to insert the new node, either as a child or at the end of the root list, then the relevant CAS succeeds, by the assumption about non-interference. □

7 union

In this section we present a simple implementation of the union operation. Recall that we assume here that there are no concurrent operations on the giving heap (denoted "giver", below), although we do allow concurrent operations on the receiving heap (the heap of the operation); we relax this assumption in Section 11. We also assume that the two heaps are distinct.

The union operation works by locating the last root node of the current heap, and updating its next pointer to reference the first (non-header) node of giver. Most of the work is done by the findLast function (Figure 14), which finds the last root and its state, also helping with the operation of any label on the last root. The traversal is almost identical to that in insert (except without trying to insert nodes). If it helps with the deletion of the last root, it tries to continue from that node's last child, if there is one, since that node is likely to be near the end of the root list; otherwise it restarts the traversal. If it helps with a merge, it continues from the same node.

The union function is then straightforward. If the giving heap is empty, there is nothing to do. Otherwise it finds the last root of the heap, and tries to update it so its next pointer points to the first proper root of the other heap; we call this the *joining CAS*. It then clears the other heap. If the joining CAS is unsuccessful, it retries.

Lemma 39 *The union function operates correctly, assuming there are no concurrent updates on its argument giver.*

```
1  private def findLast : (Node, NodeState) = {
2    var curr = head; var currState : NodeState = null // current node, state
3    def skip(a: Node, b: Node, bState: NodeState) = {
4      val next = skipOverMerge(a, b, bState)
5      if(next == null) help(curr, currState) else curr = next
6    }
7    while(true){
8      currState = curr.maybeClearParent()
9      if(currState.parent != null) curr = currState.parent
10     else currState.label match{
11       case MergeNext(a, b, bState) => skip(a, b, bState)
12       case MergeParent(pred, _, _, b, bState) if curr == pred => skip(curr, b, bState)
13       case delLabel @ Delete(pred) =>
14         if(currState.next != null) curr = currState.next
15         else{
16           helpDelete(pred, curr, currState, delLabel)
17           val ch = curr.getState.children
18           if(ch.nonEmpty) curr = ch.last else curr = head
19         }
20       case label =>
21         if(currState.next != null) curr = currState.next
22         else if(label != null) help(curr, currState) // MergeParent label
23         else return (curr, currState)
24 } } }
25 def union(giver: ConcHeap) = {
26   val giverFirst = giver.head.getState.next
27   if(giverFirst != null){ // other heap not empty
28     var done = false
29     while(!done){
30       val (last, lastState) = findLast
31       assert(lastState.next == null && lastState.parentless && lastState.unlabelled)
32       done = last.compareAndSet(lastState, lastState[next ↦ giverFirst])
33     }
34     giver.head.next = null
35 } }
```

Figure 14: The findLast and union functions.

Proof: It is clear that, if the state of last is still lastState, it is the last root node, by Lemma 29. Hence the CAS successfully appends the nodes of giver after this heap.

It is straightforward to check that all the invariants are inherited from the two component heaps (given that the heaps are disjoint). It is also clear that the operation follows the two rules. □

Lemma 40 *Consider an execution, during which, from some time on, no other thread performs any update to any node. Then* union *terminates in a finite number of steps, outside of helping.*

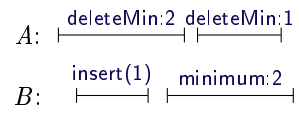
Proof: The proof is very similar to that for insert (Lemma 38): it traverses the root list in an identical way, so will eventually reach the final node; as for insert, the number of helping iterations is bounded; the final CAS must succeed. □

8 Minimum

We now discuss the minimum function, which finds the smallest key in the heap. It traverses the root list, in a similar style to findPred.

If the minimum key has been marked for deletion before the traversal completes, but has not been fully deleted, it can still be returned by minimum. We justify correctness, i.e. linearizability, in Section 13. The fact that the deletion has not been completed means that the relevant deleteMin function has not yet returned: the linearizability proof makes use of this. However, our proof of linearizability requires that the node should not have been marked before the call to minimum: we use the timestamps in the Delete labels to test this[10]. We assume that the timestamping mechanism is correct in the following sense: suppose one thread generates a timestamp ts_1 and writes it to shared memory; and a second thread later generates a second timestamp ts_2; then $ts_1 \leq ts_2$ (ignoring overflow).

It would not (in general) be correct to return a key found during the traversal if the corresponding node is fully deleted before the traversal is finished, even if the node had not been marked for deletion before the call to minimum began. Consider a heap that initially contains the keys 2 and 3, in that order, and consider the execution illustrated by the timeline to the right, and explained below.



1. Thread A starts a deleteMin, identifies 2 as the minimum key, and stalls;
2. Thread B inserts key 1 (at the end of the root list);
3. Thread B calls minimum, starts a traversal, sees the key 2, advances to 3, but then stalls;
4. Thread A resumes, fully deletes the key 2, and returns;
5. Thread A performs a deleteMin, and deletes 1;
6. Thread B resumes, completes the traversal, and returns 2.

The above is not linearizable: at no point during the execution of minimum is 2 the minimum key.

It can be advantageous, when traversing the root list, not to keep track of just a single smallest node[11], but instead to keep track of the m smallest nodes for some $m > 1$. If the smallest node is fully deleted before the traverse finishes, we can instead consider the next smallest node. This will be even more the case in the deleteMin operation (Section 9): when the deleteMin finishes the traversal, there is a fairly high likelihood that the smallest node has already been marked for deletion by another deleteMin operation; in this case, it can try the next smallest node.

The minimum function traverses the list. It keeps track of the smallest nodes that it has seen, up to a maximum of numMinsMinimum such, and discarding others. If, at the end of the traversal, the smallest node minNode has been

[10] An alternative would be to use a shared counter as a form of logical timestamp.
[11] We use terms such as "smallest node" to mean the node with the smallest key.

deleted, then minimum could return the next smallest node found during the traversal, or one of minNode's children, whichever is smaller. This requires some care. If more nodes were inserted after the initial traversal finished, but before minNode was found to be deleted, then we need to take those additional nodes into account to ensure linearizability. We therefore resume the traversal from the node that was previously the last node in the root list.

We use a subsidiary class MinList, outlined to the right, to store these minimal nodes. In fact, this class is polymorphic, to allow it to be re-used in the traversal of deleteMin. It stores up to numMins elements of type N, namely those that are minimal under the getKey function. In minimum, we instantiate N with Node, and getKey with the function that returns a node's key.

```
class MinList[N](numMins: Int, getKey: N => Int){
  /** Insert node into this. */
  def insert(node: N) : Unit
  /** Get the minimum element. Returns null if
    * the minimum element has been discarded. */
  def get : N
  /** Is this MinList empty? */
  def isEmpty : Boolean
  /** Remove the minimum element.
    * Precondition: the minimum element has
    * not been discarded. */
  def removeFirst() : Unit
  /** Clear this. */
  def clear : Unit
}
```

We omit the implementation since it is straightforward (and sequential). Note that the insert operation starts to discard nodes once the limit of numMins is reached. A critical point, however, is the following: if previously a node n was discarded because the limit had been reached, and subsequently a node was removed from the MinList (creating a space) but found to be deleted, and then another node is inserted (either a child of the deleted node, or a node found during the resumed traversal), then we should not store such nodes with a larger key than a node that was previously discarded; otherwise, if *all* the nodes found during the initial traversal are subsequently found to be deleted, we could incorrectly return a key that was larger than one in a discarded node. We therefore record the minimal key that has been discarded, and do not subsequently store a node with a larger key. If a node has been discarded and no other node is stored, a call to get returns null; the traversal has to restart in this case.

The critical properties of MinList are captured by the following lemma.

Lemma 41 *If get returns a non-null node, it is the smallest node that was added to the MinList since the last call of clear and has not been removed. isEmpty returns true if and only if every node that was added to the MinList since the last call of clear has also been removed. If get returns null, then there have been at least numMins previous calls to get since the last call of clear.*

The minimum function (Figure 15) traverses the list, keeping track of the minimal nodes encountered in minList. At each step, it uses advance to obtain the next node and its sequence number, and maybe some nodes skipped over. If advance returns null, minimum restarts. Otherwise, it inserts relevant nodes into minList; if a node skipped over is marked for deletion, it is inserted only if the marking was after the start of the traversal; otherwise the children are recursively inserted.

When the traversal reaches the end of the root list, if minList is empty, then the heap was empty, so minimum returns None (line 16). If the call to minList.get returns null, then the minimum node has been discarded, so the traversal has to restart (line 19). Otherwise, if the minimal node minNode has not been

```
1  def minimum : Option[Int] = {
2    val minList = new MinList[Node](numMinsMin, _.key)
3    var curr = head; var currSeq = curr.getSeq; val startTime = java.lang.System.nanoTime
4    def restart() = { curr = head; currSeq = curr.getSeq; minList.clear }
5    def insertNodes(ns: List[Node]) = for(n <- ns){
6      val nState = n.getState
7      nState.label match{
8        case delLabel: Delete if delLabel.ts - startTime <= 0 => insertNodes(nState.children)
9        case _ => minList.insert(n)
10   } }
11   while(true) advance(curr, currSeq) match{
12     case null => restart()
13     case (next, nextSeq, skipNodes) =>
14       insertNodes(skipNodes)
15       if(next != null){ minList.insert(next); curr = next; currSeq = nextSeq }
16       else if(minList.isEmpty){ assert(curr == head); return None }
17       else{ // at end of list; find first non-deleted node in minList
18         val minNode = minList.get
19         if(minNode == null) restart()
20         else if(! minNode.deleted) return Some(minNode.key)
21         else{ minList.removeFirst; insertNodes(minNode.getState.children) }
22 } }   }
```

Figure 15: The minimum function.

fully deleted, minimum returns minNode's key. If minNode has been fully deleted, minNode is replaced by its children in minList.

We want to prove that all the nodes in the heap are represented by the nodes added to minList. More precisely, each node in the heap is a *descendant* of one of those nodes.

Definition 42 *We define the* children *of node n to be $n.children$, unless n is the a node of a merge, and the b update has taken place, in which case we also include b within the children (recall that we consider the merge to take place at the b update).*

- *If n has a $MergeParent(\_,\_,\_,b,\_)$ label, and $b.parent = n$, then $children(n) = n.children \cup \{b\}$;*
- *Otherwise $children(n) = n.children$.*

We define the descendants *of a node n to be n itself, its children, its children's children, and so on. Inductively:*

$$descendants(n) \quad = \quad \{n\} \cup \bigcup \{descendants(c) \mid c \in children(n)\}.$$

We lift descendants to sets of nodes, pointwise:

$$descendants(S) = \bigcup \{descendants(n) \mid n \in S\}.$$

We write $nodesUpTo(r)$ for the descendants of the roots up to r:

$$nodesUpTo(r) \quad = \quad descendants(rootsUpTo(r)),$$

(recall Definition 23) so $nodesUpTo(null)$ represents all the nodes in the heap. We write $unmarkedNodesUpTo(r)$ for the nodes in $nodesUpTo(r)$ that do not have a Delete label.

Part of the invariant of the traversal is that the unmarked nodes up to the current position are a subset of the descendants of nodes added to minList. The following lemma identifies circumstances under which this property is maintained by actions of other threads.

Lemma 43 *Let S be a set of nodes. Suppose root node r does not have a Delete label, and suppose*

$$unmarkedNodesUpTo(r) \quad \subseteq \quad descendants(S). \tag{1}$$

Then equation (1) is maintained by each step of an operation that does not change the sequence number of r, other than a giving union.

Proof: Note that, by Lemma 19, r remains a root after the update. Hence $unmarkedNodesUpTo(r)$ is still well defined.

Suppose the insert operation inserts a new node n below a root r_1 no later than r; then n is added to both sides of (1): previously r_1 was a member of the right-hand side, so a descendant of a member of S; subsequently, n is a descendant of that same member of S.

If insert adds the new node below a later root, or at the end of the root list, then this does not change $unmarkedNodesUpTo(r)$, and does not reduce the right hand side of (1).

Consider the marking for deletion of a node dn. Necessarily $dn \neq r$, since this step changes dn's sequence number. The step removes dn from the left-hand side of (1) if dn is before r. It does not change the descendants of dn, so doesn't change the right-hand side.

Consider the pred update of a deletion of node dn, promoting its children to roots; necessarily, $dn \neq r$ (since dn has a Delete label). It maintains both sides of (1): in particular, if dn is before r, the descendants of dn remain in both sides.

Consider the merger of two trees, a and b, in particular the b update. Note that $r \neq$ b, since the sequence number of b is increased by this step.

- If b is before r in the root list, and a is after r, then this removes $descendants$(b) from the left-hand side of (1), but does not change the right-hand side.
- Suppose b is after r in the root list, and a is before or equal to r; then this adds $descendants$(b) to the left-hand side of (1); but previously a was a member of the right-hand side, so was a descendant of a node $n \in S$; but subsequently the descendants of b are also descendants of n, so this update also adds them to the right-hand side of (1).
- In other cases, the left-hand side of (1) doesn't change, and no node is removed from the right-hand side.

No other step changes either side of (1). □

The following proposition gives a correctness condition for minimum. In Section 13, we use this result to prove linearizability. Below, by "the end of the traversal" we mean, more precisely, when the final node's state was read in advance (line 4 of Figure 10), i.e., the state whose next field was subsequently found to be null.

Proposition 44 *If minimum returns a key k then, at the end of the traversal, the corresponding node was not deleted, and there was no smaller unmarked node in the heap. If minimum returns None, then at some point during the call,*[12] *the heap had no unmarked node.*

Proof: As in Lemma 25, if the sequence number of curr has not changed, i.e. curr.getSeq = currSeq, then curr is a root node, and does not have a Delete label.

Let S be the set of nodes that have been added to minList, but not removed, since the last time it was cleared. We show that the following is invariant: if the sequence number of curr has not changed, then

$$unmarkedNodesUpTo(\mathsf{curr}) \quad \subseteq \quad descendants(S). \tag{2}$$

This is true initially, and is re-established whenever the traversal restarts.

Consider a call to advance that returns a result (next, nextSeq, skipNodes), in an iteration that neither returns a value nor restarts the traversal. By part 2 of Lemma 21,

$$nodesUpTo(\mathsf{next}) = \\ nodesUpTo(\mathsf{curr}) \cup descendants(\mathsf{skipNodes}) \cup descendants(\mathsf{next}).$$

The additions to minList have the effect of adding (at least) the unmarked nodes of $descendants(\mathsf{skipNodes}) \cup descendants(\mathsf{next})$ to the right-hand side of equation (2), thereby maintaining this equation when curr is set to next.

If, at line 21, minNode is removed from minList, and its children added, then this removes just minNode from the right-hand side of (2). But minNode was marked for deletion, so this maintains the property.

Equation (2) is maintained by steps of other threads, by Lemma 43. Hence this invariant is maintained.

Now suppose advance returns Some(minNode.key) at line 20. Then by Lemma 41, this value was minimal in S. Hence, by equation (2) and Invariant 1, there was no smaller unmarked node in the heap. Further minNode is not deleted.

Finally suppose advance returns None at line 16. Then minList was empty, so by Lemma 41, $S = \{\}$. Hence, by equation (2), there is no unmarked key in the heap. □

The following lemma will prove useful in proving linearizability; it follows from the use of timestamps, and the fact that we assume that a deleteMin operation can not be concurrent to a giving union.

Lemma 45 *If a minimum operation returns the key from node n, then n was not marked for deletion when the operation started its traversal; further, it was not marked for deletion while in a different heap.*

The following lemma captures a liveness property for minimum.

Lemma 46 *Suppose a call to minimum occurs such that, from some point on, no other thread makes any update to any node. Then the call returns after a finite number of steps.*

[12] In fact, this property holds at the end of the traversal; however, in Section 11 we will have the slightly weaker property stated here; and this weaker property is enough for our proof of linearizability.

Proof: The proof that the main loop eventually terminates, reaching the end of the root list, is essentially identical to that for findPred (Lemma 26). If a traversal runs after all interference ends, minNode cannot have been deleted in the meantime, so minimum returns. □

9 deleteMin

We now describe the traversal used within deleteMin; see Figure 16. We need to identify the minimum node and its predecessor. The code is very similar to the code for minimum. minList stores up to numMinsDelMin pairs of the form (p, n), where n is a node encountered during the traversal, and p is its expected predecessor; these pairs are ordered by the key of the n components. Our experiments suggest a value of around 16 works well for numMinsDelMin; this roughly doubles throughput over recording a single minimum. (We expect that with more threads, a higher value would work better: we leave experiments investigating this as future work.)

While traversing, deleteMin inserts each node and its predecessor into minList (line 21). Similarly, it iterates through the nodes skipped over, and through the children of such nodes marked for deletion (via the helper function insertChildren), and inserts each and its predecessor into minList. (Recall that the correctness of the predecessor is not essential for the subsequent deletion, but is a useful optimisation.)

When it reaches the end of the list, if minList is empty, then the heap is empty, so None is returned. If the call to minList.get returns null, then the minimum node has been discarded, so the traversal restarts. Otherwise, it extracts the minimal node minNode and its expected predecessor predMin from minList. It passes these to the function tryDelete (see below), which carries out various checks to test whether the deletion might succeed, maybe helping with other operations, and then tries to delete the node itself. If this is successful, minNode's key is returned. If the deletion failed because minNode has been marked for deletion by another thread, minNode is replaced in minList by its children. If the deletion failed for some other reason, it is necessary to restart the traversal.

The most likely cause of the deletion failing is if another thread has already claimed the node for deletion. tryDelete therefore tests this first (line 39). If delNode has a label other than a Delete, then tryDelete helps with that operation (line 40), and then retries. If delNode was found as the child of a node without a Delete label, then it must have been merged below another node since it was encountered in the traversal; it cannot now be deleted so tryDelete fails. Otherwise, it calls either delete or deleteWithParent, as appropriate, to try to delete the node, recursing if unsuccessful.

The proof of the following result is nearly identical to that of Proposition 44, so we just sketch it.

Proposition 47 *If deleteMin returns Some(k) then k was a minimal unmarked key when the traversal ended. If deleteMin returns None then the heap had no unmarked nodes at some point during the call.*

Proof: (sketch). The main loop has essentially the same invariant as minimum, except referring to just the second components of the elements of minList. Hence if minList is empty at line 22, the heap is again empty. Otherwise, at line 24,

```
def deleteMin : Option[Int] = {
  val minList = new MinList[(Node,Node)](numMinsDelMin, _._2.key)
  def insertChildren(pred: Node, children: List[Node]) = {
    var cs = children; var p = pred
    while(cs.nonEmpty){ val c = cs.head; cs = cs.tail; minList.insert(p, c); p = c }
  }
  var curr = head; var currSeq = curr.getSeq // current node and seq. no.
  def restart() = { Backoff(); curr = head; currSeq = curr.getSeq; minList.clear }
  while(true) advance(curr, currSeq) match{
    case null => restart()
    case (next, nextSeq, skipNodes) =>
      var pred = curr
      for(skipNode <- skipNodes){
        val skipState = skipNode.getState
        if(skipState.label.isInstanceOf[Delete]){
          val children = skipState.children; insertChildren(pred, children)
          if(children.nonEmpty) pred = children.last
        }
        else{ minList.insert(pred, skipNode); pred = skipNode }
      } // end of for loop
      if(next != null){ minList.insert(pred, next); curr = next; currSeq = nextSeq }
      else if(minList.isEmpty){ assert(curr == head); return None }
      else{
        val minPair = minList.get
        if(minPair == null) restart()
        else{
          val (predMin, delNode) = minPair
          if(tryDelete(predMin, delNode)) return Some(delNode.key)
          else{
            val delState = delNode.getState
            delState.label match{
              case delLabel @ Delete(pred) =>
                minList.removeFirst; insertChildren(pred, delState.children)
              case _ => restart()
} }   } } } }
private def tryDelete(pred: Node, delNode: Node): Boolean = {
  val delState = delNode.maybeClearParent(); val label = delState.label
  if(label != null)
    if(label.isInstanceOf[Delete]) false // fast return in this case
    else{ help(delNode, delState); tryDelete(pred, delNode) }
  else{
    val parent = delState.parent
    if(parent != null){
      val pState = parent.getState
      pState.label match{
        case Delete(pPred) =>
          deleteWithParent(pred, delNode, delState, pPred, parent, pState) ||
            tryDelete(pred, delNode)
        case _ => false  // minNode not a root, so retry
      }
    }
    else delete(pred, delNode, delState)) || tryDelete(pred, delNode)
} }
```

Figure 16: The deleteMin and tryDelete functions

there is again no smaller unmarked node than minNode; and the deletion succeeds only if minNode is not already marked for deletion, as required. □

We now prove a liveness result for deleteMin. We start by considering tryDelete.

Lemma 48 *Suppose a thread repeatedly calls tryDelete with the same value of delNode, such that delNode is not already marked for deletion. Suppose further that no other thread makes any change to a node. Then the deletion performs a finite number of steps outside of helping.*

Proof: By assumption, delNode was not marked for deletion, and cannot have been labelled subsequently, so tryDelete does not return at line 39.

If delNode has a label other than Delete, tryDelete helps with that operation (line 40). By Lemma 8, only a bounded number of calls will be needed to complete that operation, and each instance will terminate in a finite number of steps.

Otherwise, tryDelete calls delete or deleteWithParent. This call must complete in a finite number of steps, outside of helping, by Proposition 27. In particular, since delNode is unlabelled, the deletion must succeed. □

Lemma 49 *Suppose a call to deleteMin occurs such that, from some point on, no other thread makes an update to any node. Then the call returns after a finite number of steps, outside of helping.*

Proof: The proof that the traversal terminates is again essentially identical to that for findPred (Lemma 26). This leads to deleteMin either terminating, if the heap is empty, or calling tryDelete.

If the call to tryDelete fails, then the traversal will continue or restart. This will lead to repeated calls of tryDelete, each on one of the root nodes. (In practice, all the calls to tryDelete will use the same delNode; however, a different, nondeterministic implementation of MinList might lead to calls with different values of delNode, all with the same key.) There is a fixed finite set of root nodes, so one node will be passed to tryDelete repeatedly. Hence, by Lemma 48, eventually one of these calls will succeed, and deleteMin will return. □

10 Tidying

In this section we explain when and how threads attempt to merge trees. We implement a function tidy() that performs this merging. There is a trade-off concerning how often tidy is called. Calling it more often leads to a shorter root list, which makes traversing faster. However, tidying takes time, and also risks interfering with other operations. We therefore arrange for a thread to call tidy only occasionally. More precisely, we use a shared integer counter opCount,[13] and an integer parameter tidyRatio, indicating how frequently a thread should perform tidying. Every time a thread completes an insertion or deletion, it calls the function maybeTidy (Figure 17). This increments opCount; if it is then a multiple of tidyRatio, it calls tidy. Our experiments suggests a value of between 2 and 8 for tidyRatio works best: slightly different values are optimal in different use

[13] An alternative would be to use thread-local counters.

```
private val opCount = new AtomicInteger(0)
private def maybeTidy() = if(opCount.incrementAndGet%tidyRatio == 0) tidy()
```

Figure 17: The maybeTidy function.

cases, but the implementation is fairly robust to small variations. In addition, when a thread completes a union operation, it calls tidy directly (since it is likely that this operation has significantly lengthened the root list).

The basic idea of the tidy function (Figure 18) is to traverse the root list, to identify pairs of root nodes with the same degree that could be merged. It maintains an array buckets of pairs of nodes. If buckets(d) holds a pair ($pred$, n) then, when these nodes were encountered earlier, n was a node of degree d and had predecessor $pred$, or $pred$ was likely to become the predecessor of n after some pending operation completed. If buckets(d) = null, then no such pair is known about. (For simplicity, we assume an upper bound MaxDegree on the degrees that nodes can have.)

As an optimisation, we do not perform a merge of the last root node: insert operations use this node, so merging such a node can disrupt insertions. In addition, we avoid merging singleton nodes, but mainly depend upon insert to add nodes below them: again this avoids disrupting insertions. However, in unusual cases —such as keys being inserted in decreasing order— it *is* necessary to merge singleton nodes; our approach is to try merging such nodes with probability $\frac{1}{4}$.

Most of the work is done by the function tryMerge (Figure 18). This takes a pair of nodes (pred, curr), with pred expected to be the predecessor of curr. If there is no pair of nodes stored corresponding to the degree of curr, it stores this pair (line 26). If there is a pair (predOther, other) in buckets such that other has the same degree, and the relevant preconditions for merging hold, then it attempts to merge them (lines 18 and 22). If a relevant precondition does not hold, then various heuristics are used, possibly to replace (predOther, other) in buckets with (pred, curr).

tidy itself uses a helper function nextStep (Figure 18) to find the next node next after curr to consider. nextStep also returns a boolean to indicate if next is likely to be the successor of curr (possibly after some pending operation has finished), and so this pair of nodes can usefully be passed to tryMerge. The nextStep function is similar to, but simpler than, an iteration of insert: it skips over nodes that are about to be merged below another node. It is simpler than insert, in that when it reaches the end of the list, it simply returns null for next.

The following lemma is easy to check.

Lemma 50 *Each call to merge from tidy satisfies the preconditions of merge.*

Lemma 51 *If tidy runs, and, from some point on, no changes are made to the root list by other threads, then it performs a finite number of steps.*

Proof: Recall (Lemma 7) that each call to merge terminates in a finite number of steps.

We can show that the main loop of tidy performs a finite number of iterations. The proof is very similar to (but simpler than) the corresponding proof for insert (Lemma 38), so we just sketch it. In most cases, tidy progresses along the root

```
1  private def tidy() = {
2    val buckets = new Array[(Node,Node)](MaxDegree)
3    def tryMerge(pred: Node, curr: Node) : Unit = {
4      val currState = curr.maybeClearParent; val degree = currState.degree
5      if(currState.next != null && (degree != 0 || random.nextInt(4) == 0)){
6        if(buckets(degree) != null){
7          val predState = pred.maybeClearParent; val (predOther, other) = buckets(degree)
8          val otherState = other.maybeClearParent;
9          val predOtherState = predOther.maybeClearParent
10         if(other == curr || !(otherState.unlabelled && otherState.parentless)){
11           if(predState.next == curr) buckets(degree) = (pred, curr)
12         }
13         else if(otherState.degree != degree)
14           buckets(degree) = if(predState.next == curr) (pred, curr) else null
15         else if(currState.unlabelled && currState.parentless){ // try merging
16           if(other.key <= curr.key){
17             if(predState.next == curr && predState.unlabelled && predState.parentless)
18               merge(other, otherState, pred, predState, curr, currState)
19           }
20           else if(predOtherState.next == other && predOtherState.unlabelled &&
21                   predOtherState.parentless)
22             merge(curr, currState, predOther, predOtherState, other, otherState)
23           else if(predState.next == curr) buckets(degree) = (pred, curr)
24         }
25       } // end of if(buckets(degree) != null)
26       else if(predState.next == curr) buckets(degree) = (pred, curr)
27   } } // end of tryMerge
28   var curr = head; var done = false
29   while(!done){
30     val (next, ok) = nextStep(curr)
31     if(next == null) done = true
32     else if(ok){ val pred = curr; curr = next; tryMerge(pred, curr) }
33     else curr = next
34 } }
35 private def nextStep(curr: Node) : (Node, Boolean) = {
36   def skip(a: Node, b: Node, bState: NodeState) : (Node, Boolean) = {
37     val myBState = b.getState
38     if(myBState == bState || myBState.parent == a) (bState.next, true) else (b, true)
39   }
40   val currState = curr.maybeClearParent()
41   if(currState.parent != null) (currState.parent, false)
42   else currState.label match{
43     case MergeNext(a, b, bState) => skip(a, b, bState)
44     case MergeParent(pred,_,_,b,bState) if curr == pred => skip(curr, b, bState)
45     case Delete(_) => (currState.next, false)
46     case _ => (currState.next, true)
47 } }
```

Figure 18: The tidy function.

list in the same way as for a step of insert; when tidy reaches the end of the list, it terminates (whereas insert sometimes helps). Thus tidy takes no more iterations than insert from the same node, i.e. a finite number. □

We now sketch a small enhancement to tryMerge. If a call to merge appears to be successful, then tryMerge can recurse with the new root and its expected predecessor. Similarly, if the degree of other has changed, it can try recursing on other.

11 Allowing concurrent unions

In this section we explain how to re-implement the union operation so as to allow it to run concurrently with other operations on the giving heap. We start by highlighting some of the difficulties, so as to motivate parts of our design; in particular, we show what could go wrong with a less careful implementation.

Suppose a call to minimum occurs on heap h_g concurrently to a call to h_r.union(h_g). Suppose, further, that immediately after the two heaps are joined, a new key k is added to h_r. Then the traversal of minimum may encounter k, and incorrectly return it, even though it is not part of h_g. We avoid this problem as follows: when the traversal finishes, the thread detects whether a giving union on h_g has been linearized during the traversal; if so, minimum returns None, and the call can be linearized immediately after the linearization point of union, at which point the heap was indeed empty. The main mechanism we use to detect whether such a union has been linearized is to equip each heap with an integer variable epoch which counts the number of such unions so far.

A similar problem arises with deleteMin. However, the above approach is not sufficient. Consider the following behaviour. (1) Thread A calls deleteMin on heap h_g, identifies k as the minimum key, checks that no concurrent union has occurred, but then is suspended; (2) thread B performs h_r.union(h_g); (3) thread C calls minimum on h_r, and returns k; (4) thread A resumes and deletes k. This behaviour is not linearizable because of the delay between the check in step 1 and the update in step 4: in effect, thread A has deleted k from h_r rather than h_g. To avoid this, we ensure that no giving union occurs concurrently to the delNode labelling of a deletion. We explain the mechanism below.

Further, suppose deleteMin labels a node *delMin* for deletion, but has to call findPred to locate the predecessor of *delMin*. If *delNode* has been moved to a different heap in the meantime, then obviously findPred has to search in that different heap. We therefore need a mechanism for locating the current heap that a node is in. Also, if findPred fails to find *delNode*, there are subtleties in deciding whether it has really been decoupled so can be marked as fully deleted.

Similarly, without care, unions can interfere with insertions on the giving heap, so that the insertion happens after the union, and so effectively inserts into the receiving heap.

Finally, two concurrent unions on the same heap could interfere with one another. Consider two threads performing h_1.union(h_2) and h_2.union(h_3). Suppose the former joins its two heaps first. Then, without suitable care, the latter thread could join h_3 onto the combined heap, h_1, rather than h_2. Similarly, if we have a cycle of unions, for example if $h_1 = h_3$, above, then this could create a circular root list. We prevent this by labelling both heaps, more precisely their head nodes, at the start of the union, with UnionGiver and UnionReceiver labels. This prevents concurrent unions affecting the same heap. However, we clearly need a mechanism to detect a cycle of such labelling, which could create a deadlock.

Similarly, to prevent a giving union concurrently to the critical step of an insertion or deletion (as justified above), we add a special label, which we call a *heap label* to the head node during that critical step. However, we can allow the critical steps of insertions or deletions to occur concurrently to one another, or concurrently to the heap being the receiving heap of a union. We therefore allow the head node to have a *list* of heap labels and at most one UnionReceiver

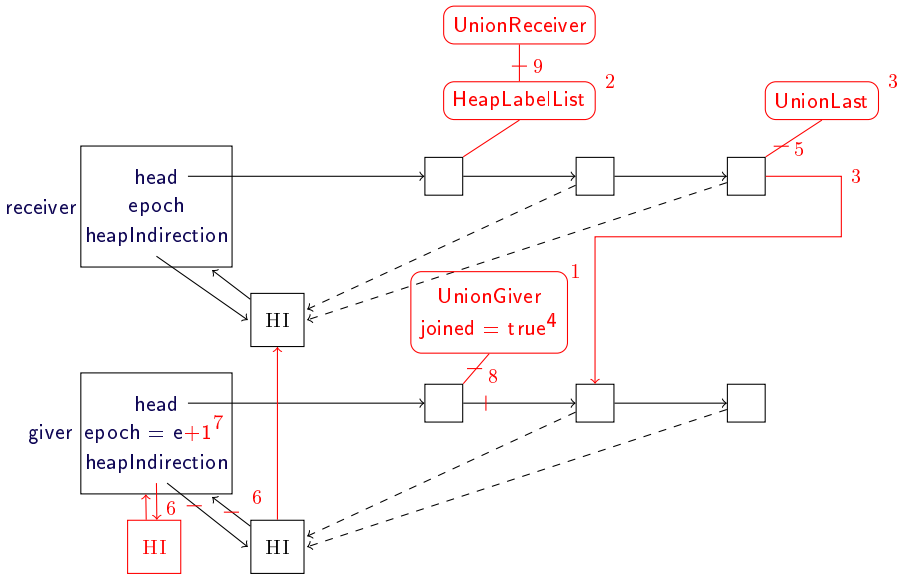


Figure 19: Illustration of union. Drawing conventions extend those in Figure 2 as follows: the large boxes on the left are heap objects; the boxes labelled "HI" are heap indirections; dashed lines from a node to a heap indirection represents a path via zero of more intermediate heap indirections.

label, allowing these to occur concurrently, but not concurrently to a giving union; these are encapsulated in a HeapLabelList.

Recall that we need a mechanism for enabling a thread to identify the current heap that a node is in; we want to be able to update this mechanism efficiently when a union happens. We use a technique based upon the union-find data structure [GF64]. Each heap has a *heap indirection* object. Each heap indirection contains a reference to either its heap or another heap indirection object. When a node (other than the dummy header) is added to a heap, it is given a reference to the corresponding heap indirection. Thus following a path of such references leads to the corresponding heap, except temporarily during a union. Each union operation updates the heap indirections to re-establish this property.

We now outline how union proceeds. Figure 19 illustrates the operation.

1. If the giving heap is empty, the union operation is equivalent to a no-op, and so returns immediately. Otherwise, it labels the head node of the giving heap with a UnionGiver label. This has a side effect of ensuring that the first proper node remains as such. The UnionGiver label acts as a convenient place to store information about the state of the union.

2. It tries to label the head node of the receiving heap with a UnionReceiver label, either adding a HeapLabelList to hold it, or using an existing HeapLabelList. If this addition is blocked because that node already has a UnionGiver label, it tests whether there is a cycle of such labelled nodes, and if so removes one such label and replaces it with the corresponding UnionReceiver

label; the union operation corresponding to the replaced label then restarts.

3. It finds the last node in the receiving heap, and updates its next pointer to point to the first proper node of the giving heap. We call this the *joining CAS*. It is the linearization point of the union: we consider the nodes to be transferred from the giving heap to the receiving heap at this point. If the receiving heap is non-empty, it simultaneously labels that last node with a UnionLast label; this allows other threads to detect that the linearization point has passed. (If the receiving heap is empty, the last node is the head, and retains its HeapLabelList label.)

4. It updates the UnionGiver label to indicate that the joining CAS has occurred.

5. If the receiving heap was non-empty, it removes the UnionLast label from the former last node of the receiving heap.

6. It sets the giving heap's heap indirection to reference that of the receiving heap; hence following the path of heap indirections from a former node of the giving heap leads to the receiving heap, as required. The giving heap receives a new heap indirection.

7. It increments the giving heap's epoch.

8. It atomically sets the giving heap's head's next reference to null and removes the UnionGiver label.

9. It marks the UnionReceiver label as complete, and removes it from the HeapLabelList.

As with previous operations, other threads might help with steps after the first; and it might be necessary to help other operations that block the union.

Each UnionGiver and HeapLabelList applies to the whole heap, rather than to an individual node. In a previous version, we made them attributes of the heap, rather than the head node. However, it also proved necessary to label the head of the giving heap, to prevent the reference to the first node from changing before the giving CAS: it simplified things to combine these two labels.

We augment the state of each heap with an epoch and the heap indirection.

```
private val epoch = new AtomicLong(0)
private val heapIndirection = new AtomicReference(new HeapIndirection(this))
```

We describe the HeapIndirection class later.

In the remainder of this section, we describe the union operation in more detail: we start by outlining HeapLabelLists; in the following four subsections, we describe the different phases of the union; we then describe how other threads help with the union, and give correctness results. In the next section we describe how other operations have to be adapted to allow concurrent giving unions.

11.1 HeapLabelLists

We now sketch the implementation of HeapLabelLists. Recall that each HeapLabelList holds a list of HeapLabels, and at most one UnionReceiverLabel.

```
1  class HeapLabel{
2    /** The next label in the list. */
3    val next = new AtomicReference[HeapLabel](null)
4    /** Has this operation been completed (perhaps unsuccessfully)? */
5    @volatile var done = false
6  }
7  /** A thread is trying to insert node below parent. */
8  case class InsertBelow(node: Node, parent: Node, parentState: NodeState, epoch: Long)
9    extends HeapLabel
10 /** A thread is trying to insert node after pred. */
11 case class InsertAtEnd(node: Node, pred: Node, predState: NodeState, epoch: Long)
12   extends HeapLabel{
13   /** Has the insertion happened successfully? */
14   @volatile var success = false
15 }
16 /** A thread is trying to label delNode for deletion. */
17 case class LabelForDelete(delNode: Node, delState: NodeState, newNext: Node,
18                           startEpoch: Long, delLabel: Delete) extends HeapLabel
19 /** This heap is acting as the receiver in a union. */
20 class UnionReceiver(val giver: Union){
21   @volatile var done = false // Has the union been completed?
22 }
```

Figure 20: HeapLabels.

```
1  case class HeapLabelList private (heap: BinomialHeap) extends Label{
2    /** Constructor to initialise with a UnionReceiver. */
3    def this(heap: BinomialHeap, ur: UnionReceiver)
4    /** Constructor to initialise with a HeapLabel. */
5    def this(heap: BinomialHeap, hl: HeapLabel)
6    /** Add hl to the list. */
7    def add(hl: HeapLabel): Boolean
8    /** Add ur to this, helping to complete the current UnionReceiver if necessary. */
9    def add(ur: UnionReceiver): Boolean
10   /** Get the current UnionReceiver, possibly null. */
11   def getUR: UnionReceiver
12   /** Remove the current UnionReceiver if it is ur. */
13   def remove(ur: UnionReceiver): Unit
14   /** Prevent more labels from being added to this. Help to complete all the
15     * existing labels. Then remove this from head. */
16   def close: Unit
17   /** Remove done HeapLabels from the front of the list. */
18   def tidy: Unit
19 }
20 private def tidyHeapLabelList =
21   head.getLabel match{ case hll: HeapLabelList => hll.tidy; case _ => { } }
```

Figure 21: Interface of the HeapLabelList class, and the tidyHeapLabelList function.

HeapLabels are defined in Figure 20. Each contains a next reference to allow them to be formed into a linked list, and a boolean done, which is set when the relevant steps are completed. We explain the subclasses when we explain how the critical steps are implemented. Figure 20 also defines the UnionReceiver labels. These simply contain a reference to the giving heap, and a boolean indicating whether the union has been completed.

Figure 21 gives the interface of the HeapLabelList class. We only sketch the implementation, in the interests of brevity, since it is fairly straightforward. The HeapLabels are arranged in a linked list, closely following the lock-free queue of Michael and Scott [MS96]. The head of this list and the current UnionReceiver (possibly null) are encapsulated into a pair to make them atomically updateable. The two add operations add a HeapLabel or a UnionReceiver, respectively; the latter might need to help to complete the current UnionReceiver, first, but takes no action if ur is the current UnionReceiver, or if ur.done is true. The close operation helps to complete all the current operations, and then removes the HeapLabelList from the heap's head. If an add operation finds that the HeapLabelList is being closed, it helps with that and returns false to indicate a failure. The remove operation removes the current UnionReceiver; this happens *after* it is marked as done. When an operation is completed, the label's done label is set; the function tidy (as an optimization) removes such labels from the list; the function tidyHeapLabelList (in the BinomialHeap class) invokes tidy on the current HeapLabelList, if any.

Rule 4 *A node with a HeapLabelList label can be updated only as follows.*

- *To remove the label if all the operations in it are complete;*
- *To update its next reference from null, either to insert a new node, or as the joining CAS of a union;*
- *To perform the pred update of the deletion of the following node n; however if the HeapLabelList contains a UnionReceiver, and the joining CAS has happened with n as the first node of the giving heap, then the corresponding UnionGiver must first be updated to record that the CAS has happened (corresponding to step 4 of the union).*

11.2 The labelling of the giving heap

Recall that the union operation operates on the receiving heap, and takes the giving heap as an argument. However, some steps are performed on the giving heap.

The union operation starts by labelling the giving heap's head node, via the labelGiverForUnion function (Figure 22), which returns the label added and the first proper node of the giving heap. It first prepares the UnionGiver label for the giving heap, and the UnionReceiver label for the receiving heap. Figures 23 and 27 give the UnionGiver class; the class encapsulates a lot of functionality, which we explain as we use it.

The labelGiverForUnion function tests whether the giving heap is empty; if so, it returns null as the first node of the heap, and the union function returns. Otherwise, if the head has a null label, it tries to add the UnionGiver label; if the head's label is not null, it helps with the corresponding operation, and re-tries. It then initialises the UnionGiver to store the first node, the heap indirection and the epoch of the giving heap. It is important that this happens *after* the UnionGiver label is added to the header, to ensure the information is current. The information remains current until updated as part of this operation, or this labelling is backtracked, as captured by the following rule.

Rule 5 *The heap indirection and epoch of a heap change only when that heap is labelled with a UnionGiver label (at steps 6 and 7, respectively).*

```
1  def union(giver: BinomialHeap) = {
2    val (ug, giverFirst) = giver.labelGiverForUnion(this)
3    if(giverFirst != null) unionLabelReceiver(giver, giverFirst, ug, false)
4  }
5  private def labelGiverForUnion(receiver: Union): (UnionGiver, Node) = {
6    val ug = UnionGiver(receiver, new UnionReceiver(this))
7    var first : Node = null; var done = false // first will be first node
8    while(!done){
9      val headState = head.getState
10     if(headState.next == null) done = true // Empty heap
11     else if(headState.label == null){
12       if(head.compareAndSet(headState, headState.addLabel(ug))){
13         first = headState.next; initUG(ug, first); done = true
14     } }
15     else help(head, headState)
16   }
17   (ug, first)
18 }
19 private def initUG(ug: UnionGiver, first: Node) = ug.init(first, heapIndirection.get, epoch.get)
20 private def initUG(ug: UnionGiver) = initUG(ug, head.getState.next)
```

Figure 22: The union, labelGiverForUnion and initUG functions.

```
1  case class UnionGiver(receiver: Union, ur: UnionReceiver) extends Label{
2    /** A triple of the first node, the heap indirection and the epoch of the giving heap. */
3    private val state = new AtomicReference[(Node, HeapIndirection, Long)](null)
4
5    /** Initialise with the first node, heap indirection and epoch of the giving heap. */
6    def init(first : Node, hi: HeapIndirection, epoch: Long) =
7      if(state.get != null || ! state.compareAndSet(null, (first, hi, epoch)))
8        assert(state.get == (first, hi, epoch) || joined || aborted)
9    def isInit = state.get != null
10   /** The first proper node of the giving heap. */
11   def first = state.get._1
12   /** The HeapIndirection of the giving heap. */
13   def heapIndirection = state.get._2
14   /** The epoch of the giving heap. */
15   def epoch = state.get._3
16   /** If the labelling using this is aborted, the UnionReceiver label that replaces it. */
17   @volatile var replacedBy: UnionReceiver = null
18   /** Abort this labelling, setting ur to be the UnionReceiver label that replaces it. */
19   def abort(ur: UnionReceiver) = {
20     assert(replacedBy == null || (replacedBy eq ur)); replacedBy = ur
21   }
22   /** Has the labelling with this been aborted? */
23   def aborted = replacedBy != null
24   /** A sequence counter. */
25   val seq = UnionGiver.seq.getAndIncrement
26   ...
27 }
28 /** Companion object for UnionGiver. */
29 object UnionGiver{ val seq = new AtomicInteger(0) }
```

Figure 23: Part of the UnionGiver class, and the UnionGiver companion object.

```
1  private def unionLabelReceiver(
2      giver: Union, giverFirst: Node, ug: UnionGiver, helping: Boolean) = {
3    assert(((giver.head.getLabel eq ug) || ug.aborted || ug.joined) && ug.isInit)
4    var done = false; val ur: UnionReceiver = ug.ur
5    while(!done){
6      val headState = head.getState
7      if(ug.aborted){ if(!helping) return union(giver) else return }
8      else if(ur.done) return
9      else headState.label match{
10       case null  =>
11         val hll = new HeapLabelList(this, ur)
12         done = head.compareAndSet(headState, headState[label ↦ hll])
13       case hll: HeapLabelList => done = hll.add(ur)
14       case ug1: UnionGiver =>
15         breakLoop; if(!ug1.aborted && !ug.aborted) help(head, headState)
16       case _ => help(head, headState)
17     } // end of match
18   } // end of while(!done)
19   unionJoin(giver, giverFirst, ug)
20 }
```

Figure 24: The unionLabelReceiver function.

Another thread might help with the initialisation via one of the initUG functions; in each case, it provides consistent values until the joining CAS or the labelling is backtracked, as captured by the assertion within init.

Rule 6 *A node with a UnionGiver label can be updated only as follows.*

- *To clear this label and its next reference as step 8 of the union; or*
- *To replace the label with a HeapLabelList containing a UnionReceiver label, to break a blocking loop (step 2).*

11.3 The labelling of the receiving heap

The unionLabelReceiver function (Figure 24) attempts to add the UnionReceiver label ur to the receiver's head node. However, it's possible that this node is already labelled with a UnionGiver label, blocking this union, and that there is a loop of such blocking; in this case, one of the UnionGiver labels is removed, aborting that previous labelling. The function assumes that the giving heap has already been labelled with ug, although the label might have been removed if it has been backtracked or the union completed, and ug has been initialised. The parameter helping is true for helping threads.

The normal operation is to add the UnionReceiver label ur to a new or existing HeapLabelList (lines 11 and 13). However, there are several corner cases.

If the UnionGiver label ug has been backtracked, as captured by its aborted function, then the union operation restarts, unless this thread is simply helping. If the union operation has been completed (by a helping thread), as captured by the done field on ur, then the function simply returns (this is necessary to avoid re-adding the label, which could lead to the union being done twice).

If the head node is labelled with another UnionGiver label ug1, then the function breakLoop (described below) searches for a loop of blocking unions, and backtracks one such label, aborting that labelling. If neither ug nor ug1 were

```
1  /** Try to detect a loop of UnionGiver labels, rolling one back if so. */
2  private def breakLoop = {
3    var heap = this; var seen = List[(BinomialHeap, UnionGiver)]()
4    var done = false; var loop = false
5    while(!done && !loop){
6      heap.head.getLabel match{
7        case ug @ UnionGiver(h, _) =>
8          val pair = (heap, ug)
9          if(seen.contains(pair)){ loop = true; seen = pair :: seen.takeWhile(_ != pair) }
10         else{ heap = h; seen ::= pair }
11       case _ => done = true
12   } }
13   if(loop && seen.forall{ case (h,ug) => h.head.getLabel eq ug }){
14     val (latest,latestUG) = seen.maxBy(_._2.seq)
15     val List((prev, prevUG)) = seen.filter(_._2.receiver == latest)
16     if(!prevUG.isInit) prev.initUG(prevUG)
17     latestUG.abort(prevUG.ur); latest.replaceLabel(latestUG, prevUG.ur)
18 } }
19 /** Replace label ug by ur. */
20 private def replaceLabel(ug: UnionGiver, ur: UnionReceiver) = {
21   val hll = new HeapLabelList(this, ur); val headState = head.getState
22   if(headState.label eq ug) head.compareAndSet(headState, headState.addLabel(hll))
23 }
```

Figure 25: The breakLoop and replaceLabel functions.

aborted, this operation helps with ug1 and re-tries. If ug is aborted, this will be detected on the next iteration. If ug1 is aborted, it will be replaced with a UnionReceiver label: if this is ur, then this thread will detect that fact on the next iteration (the call to add at line 13 will return true); if it corresponds to some third union operation, this thread will help that operation on the next iteration (if it hasn't already been completed).

Finally, if the head node is labelled with some other label, then this is helped.

The code to detect blocking loops of union operations is in Figure 25. We say that heap h is *blocked by* heap h' if h has a UnionGiver(h', ur) label (so the corresponding union operation seeks to label h' with ur), but h' also has a UnionGiver label. The function follows the path of such blocking, recording the path of heaps and corresponding labels (in reverse order) in the variable seen. If it returns to a pair that it has seen previously it has detected a potential loop; it extracts the sub-path corresponding to that loop[14]. It then checks that each of the heaps is still labelled with the previously seen UnionGiver label[15].

The choice of which UnionGiver label in the loop to backtrack is fairly arbitrary. However, we ensure that all threads select the same label. To this end, each UnionGiver label includes a sequence number seq, initialised from the companion object (see Figure 23). All threads select the label latestUG with maximum sequence number to roll back[16]. The (unique) pair (prev, prevUG) that is blocked by latestUG is found. If necessary, prevUG is initialised. Then latestUG is marked as aborted, to be replaced by the UnionReceiver label in prevUG (see Figure 23); this ensures that all other threads that encounter latestUG know it is being aborted, and know which label to replace it by; note that all threads that

[14] Here takeWhile extracts the prefix of seen before the first instance of pair.
[15] The use of forall tests whether, for each pair (h,ug) in seen, h is still labelled with ug.
[16] Using the function maxBy.

detect the loop pass in the same value for the replacement label. The replacement itself is done by replaceLabel, replacing the UnionGiver by a new HeapLabelList containing ur.

We now prove that the code is correct in the following sense: if a UnionReceiver label is added, then the corresponding UnionGiver label was previously added to the giving heap, and initialised; the UnionGiver label has not been aborted and will not be aborted subsequently; and no UnionReceiver label is added more than once. We start by considering breakLoop.

Lemma 52 *Suppose a call to breakLoop aborts a particular UnionGiver label latestUG, replacing it by a UnionReceiver label prevUG.ur. Then*

1. *latestUG.ur has not been added to a node;*
2. *prevUG has been initialised, but not aborted, at the point of replacement.*

Proof: Recall that after detecting a potential loop, breakLoop checks that each of the nodes is still labelled as previously. Hence all these labels must have been on those nodes throughout the period between the detection and the check (recall that once a label is removed, it is never added again to a node). All threads that detect that loop select the same label latestUG for aborting.

1. latestUG.ur cannot have been added to a node, since it would have been added to one of the other heaps in the loop, and UnionReceiver labels are removed *after* the corresponding UnionGiver label.
2. The code explicitly initialises prevUG (if necessary) before doing the replacement. prevUG cannot have been aborted prior to the point of latestUG being replaced: if it had, it would necessarily have been as a result of a different blocking cycle, and prevUG would have been removed, so the current blocking cycle wouldn't have been found.

□

Note, in particular, that the above lemma implies that once a node is labelled with a UnionReceiver label, the corresponding UnionGiver cannot be aborted.

Lemma 53 *Suppose a call to unionLabelReceiver is made following the stated precondition, and that the call adds the corresponding UnionReceiver label. Then*

1. *The UnionGiver label has not been aborted;*
2. *The UnionReceiver label is not added twice.*

Proof: Note that a UnionGiver label is marked as aborted *before* it is removed; at that point, the head node of the receiving heap also has a UnionGiver label ug1. Suppose, for a contradiction, that ug is aborted before ur is added. This must be after the test at line 7 of Figure 24, and so after headState is read.

- If ur were added at line 12, then ug1 must have been added to head after the read of headState, so the CAS would have failed.
- If ur were added at line 13, then again head's state must have been changed, to close and remove hll and replace it with the UnionGiver label ug1; but then the call to add would have failed (since hll has been closed).
- Lemma 52 dealt with the case of ur being added within breakLoop.

```
1  private def unionJoin(giver: BinomialHeap, giverFirst: Node, ug: UnionGiver): Unit = {
2    var done = false
3    while(!done){
4      ug.getLastInfo match{
5        case (null, null, false) => // Find last node
6          val (last, lastState) = findLast; val label = lastState.label
7          assert(lastState.next == null && lastState.parentless &&
8            (label == null || last == head && label.isInstanceOf[HeapLabelList]))
9          if(ug.setLastInfo(last, lastState)) done = updateLast(giverFirst, ug, last, lastState)
10       case (last, lastState, d) => // Another thread found last.
11         done = d || updateLast(giverFirst, ug, last, lastState)
12     } // end of match
13   } // end of while
14   clearLast(giver, ug)
15 }
16 private def updateLast(
17     giverFirst: Node, ug: UnionGiver, last: Node, lastState: NodeState): Boolean = {
18   val lastLabel = if(head == last) lastState.label else UnionLast(ug)
19   val joinedLastState = lastState[next ↦ giverFirst, label ↦ lastLabel]
20   if(last.compareAndSet(lastState, joinedLastState)){ ug.setDone(last, joinedLastState); true }
21   else if(ug.joined) true
22   else{ ug.clearLastInfo(last, lastState); false }
23 } // end of updateLast
24 case class UnionLast(ug: UnionGiver) extends Label
```

Figure 26: The unionJoin and updateLast functions, and the UnionLast class.

A similar argument shows that ur cannot be added twice. Recall that the done flag is set before removing the label from the HeapLabelList. Suppose the first addition is to a particular HeapLabelList hll_1. ur cannot be added a second time to hll_1: the call to add tests if its argument equals its current HeapLabelList or has had its done flag set. Hence hll_1 must have been removed before headState is read. But then the read of ur.done at line 8 would have given true. □

11.4 Joining the heaps

The joining CAS, and the subsequent update to the UnionGiver label, are coordinated by the function unionJoin (Figure 26). This step is made more complicated by the necessity to allow helping threads, and to ensure that two such threads do not perform successful CASes from different nodes. To this end, the UnionGiver class (Figure 27) contains a variable lastInfo containing an atomic reference to a triple of one of three forms:

1. (null, null, false), indicating that the joining CAS has not yet happened, and the last node of the receiving heap is not currently stored;

2. (last, lastState, false), indicating that last was identified as the last node of the receiving heap, with state lastState, but that the joining CAS might not yet have happened; threads may attempt the joining CAS from lastState;

3. (last, lastState, true), indicating that the joining CAS has been done upon node last, producing state lastState.

Each thread starts by reading lastInfo. If it receives a result of form 1, it finds the last node and its state, using findLast; we adapt this function slightly from

```
case class UnionGiver(receiver: Union, ur: UnionReceiver) extends Label{
  ...
  private val lastInfo = new AtomicReference[(Node, NodeState, Boolean)](null, null, false)
  /** The last node in the receiving heap (or null, if unknown), its state, and whether
    * the linearization point of the union has been reached. */
  def getLastInfo: (Node, NodeState, Boolean) = lastInfo.get
  /** Try to set information about last node of the receiving heap, if not already set. */
  def setLastInfo(last: Node, lastState: NodeState): Boolean = {
    val oldLastInfo @ (oldLast, oldLastState, done) = lastInfo.get
    oldLast == null && lastInfo.compareAndSet(oldLastInfo, (last, lastState, false)) ||
      oldLast == last && oldLastState == lastState && !done
  }
  /** Clear the information stored about the last node in the receiving heap, if
    * it corresponds to last and lastState. */
  def clearLastInfo(last: Node, lastState: NodeState) = {
    val oldLastInfo @ (oldLast, oldLastState, done) = lastInfo.get
    if(last == oldLast && lastState == oldLastState){
      assert(!done); lastInfo.compareAndSet(oldLastInfo, (null, null, false))
  } }
  /** Record that the joining CAS has been done, giving state newLastState for last. */
  def setDone(last: Node, newLastState: NodeState) = {
    val oldLastInfo @ (oldLast, oldLastState, done) = lastInfo.get
    assert(done || last == oldLast)
    if(!done) lastInfo.compareAndSet(oldLastInfo, (last, newLastState, true))
  }
  /** Has the joining CAS been performed (so that the union is linearized)? */
  def joined = {
    val (last, lastState, d) = lastInfo.get
    if(d) true
    else if(last != null){
      val newLastState = last.getState
      if(newLastState.next == first){ setDone(last, newLastState); true }
      else lastInfo.get._3
    }
    else false
  }
}
```

Figure 27: The UnionGiver class (continued).

the version in Section 7 to allow it to return a node labelled with a HeapLabelList, which will be the case when the receiving heap is empty. It then tries to update lastInfo to form 2; this fails (and returns false) if another thread has updated lastInfo to a different value. If the update is successful, it then tries to perform the CAS itself using updateLast (see below). Alternatively, if the read of lastInfo returns a node and state stored by another thread, if the CAS is not recorded as having been done, it again attempts that CAS using updateLast.

The updateLast function attempts to update the last node, setting the next reference to point to the first proper node of the giver heap, and adding a UnionLast label if the receiving heap is non-empty (or else retaining the HeapLabelList). If this succeeds, it updates lastInfo via setDone, into form 3. Recall that if the CAS succeeds, it is the linearization point for the union operation.

If the joining CAS fails, the function tests whether another thread has performed it, using the joined function on the UnionGiver label. Otherwise, it clears lastInfo, resetting it to form 1 if no subsequent update has been performed on it.

The joined function tests whether lastInfo is in form 3, indicating that setDone

```
1  private def clearLast(giver: Union, ug: UnionGiver) = {
2    val (last, lastState, true) = ug.getLastInfo
3    if(last != head) clearLastLabel(last, lastState)
4    completeUnion(giver, ug)
5  }
6  private def clearLastLabel(last: Node, lastState: NodeState) = {
7    assert(lastState.label.isInstanceOf[UnionLast])
8    last.compareAndSet(lastState, lastState.addLabel(null))
9  }
```

Figure 28: The clearLast and clearLastLabel functions.

has been called. Otherwise, if lastInfo is in form 2, it tests whether the stored last now points to the first node of the giving heap; if so it calls setDone to record this fact. If last does not point to first, it re-reads lastInfo in case, between lines 28 and 32, another thread called setDone, and subsequently the next reference of last was updated. Informally this logic is correct because last continues to point to first until after setDone is called; we justify this below.

The following rule applies to UnionLast labels.

Rule 7 *The only change allowed on a node n with a UnionLast(ug) label is to remove the label, which must be after a call to ug.setDone passing n and its state.*

The UnionLast label is removed from the former last node of the receiving heap, if applicable, using the function clearLast (Figure 28). The CAS is guaranteed to succeed unless another thread has already performed it, by Rule 7.

Lemma 54 *The joining of the heaps works correctly: at most one joining CAS succeeds, joining the last node of the receiving heap to the first node of the giving heap; and calls to joined return true precisely after this CAS.*

Proof: Figure 29 gives a state machine to illustrate the joining of the heaps. Each state is labelled with a 4-tuple (*lastInfoForm*, *lastStateCorrect*, *joined*, *lastLabelled*), where:

- *lastInfoForm* is either 1, 2 or 3, indicating the form of lastInfo.
- *lastStateCorrect* (in the cases *lastInfoForm* $\neq$ *1*) is a boolean indicating whether the state lastState stored in lastInfo matches the current state of last.
- *joined* indicates whether a successful joining CAS has occurred.
- *lastLabelled*, in the case that *joined* is true, is either L indicating that last is labelled with a UnionLast label, R indicating that such a label has been removed, or E indicating that the receiving heap was empty (so last has a HeapLabelList label).

We omit from the state diagram calls to setLastInfo and clearLastInfo that do not change lastInfo, and unsuccessful joining CASes. Note that:

- The state diagram captures the logic of the functions on lastInfo.
- setLastInfo may make *lastStateCorrect* true or false, depending on whether the state changed since the thread read it. Interference from other threads, changing the state of last, can make it false, but cannot make it true (since setLastInfo is always called with a previously read state, and NodeState objects are never re-used).

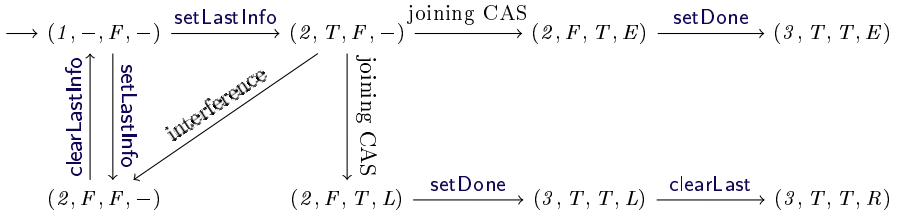


Figure 29: A state machine illustrating the joining of the heaps.

- clearLastInfo is called only after an unsuccessful joining CAS, so if it changes lastInfo, there must have been interference making *lastStateCorrect* false.
- The joining CAS can succeed only if the value of lastState used is still valid, which implies there has been no interference.
- After the joining CAS has occurred, last.next cannot change before the UnionLast label has been removed, by Rule 7, or before setDone is called, by Rule 4 (in the case of a non-empty or empty receiving heap, respectively).
- In the states described in the previous item, a call to joined returns true, so a thread whose CAS fails does not call clearLastInfo. Hence lastInfo cannot change before a call to setDone; and no subsequent joining CAS can succeed.
- After the call to setDone, subsequent calls to joined again return true; and calls to getLastInfo return a result of form 3; so again no subsequent joining CAS can succeed.
- In the case of a non-empty receiving heap, the first attempt to remove the UnionLast label succeeds, by Rule 7; and this is performed only after setDone is called.

□

11.5 Completing the union

Most of the code relating to heap indirections is in Figure 30. Each HeapIndirection object has a variable next which references either a heap, or another heap indirection[17]. Recall that each heap h has a variable heapIndirection which references a HeapIndirection; this in turn references h, except temporarily during a union. Each node has a variable heapIndirection, which is set to point to the relevant heap indirection before the node is inserted; see Section 12.1.

The getHeap operation on a node calls the getHeap operation on its HeapIndirection. This follows a path of next references until it reaches the corresponding heap. The heapOf operation on the heap normally returns that heap; however, if it is currently the giving heap in a union that has been linearized, then it returns the receiving heap. On receiving the result of heapOf, getHeap checks whether its next reference has changed in the meantime, restarting if so. Thus getHeap on a node always returns the correct heap; see Lemma 56. An

[17] Values of the type Either[BinomialHeap, HeapIndirection] are either of the form Left(h) where h is a heap, or of the form Right(hi) where hi is a HeapIndirection.

```
class HeapIndirection(heap: BinomialHeap){
  @volatile private var next: Either[BinomialHeap, HeapIndirection] = Left(heap)
  def setNext(hi: HeapIndirection) = next match{
    case Left(h) => next = Right(hi); case Right(hi1) => assert(hi1 == hi)
  }
  def getHeap: BinomialHeap = next match{
    case Left(h) => val h1 = h.heapOf; if(next == Left(h)) h1 else getHeap
    case Right(hi) => hi.getHeap
  }
}
class BinomialHeap{
  ...
  def heapOf: ConcHeap = head.getLabel match{
    case ug @ UnionGiver(receiver,_) => if(ug.joined) receiver else this
    case _ => this
  }
}
class Node{
  ...
  @volatile var heapIndirection: HeapIndirection = null
  def getHeap: BinomialHeap = heapIndirection.getHeap
}
```

Figure 30: Code relating to HeapIndirections.

```
private def completeUnion(giver: Union, ug: UnionGiver) = {
  giver.completeUnionOnGiver(ug); clearUR(head, ug.ur)
}
private def completeUnionOnGiver(ug: UnionGiver) = {
  val hi = ug.heapIndirection; val newHi = ug.receiver.heapIndirection.get
  val headState = head.getState
  if(headState.label eq ug){
    if(heapIndirection.get == hi){
      hi.setNext(newHi); heapIndirection.compareAndSet(hi, new HeapIndirection(this))
    }
    val ep = ug.epoch; if(epoch.get == ep) epoch.compareAndSet(ep, ep+1)
    head.compareAndSet(headState, headState[next ↦ null, label ↦ null])
} }
private def clearUR(node: Node, ur: UnionReceiver) = {
  ur.done = true
  node.getLabel match{ case hll: HeapLabelList => hll.remove(ur); case _ => {} }
}
```

Figure 31: The completeUnion, completeUnionOnGiver and clearUR functions.

enhancement is to perform path compression, replacing a path of next references by a single link to the last HeapIndirection on that path; we omit the details.

The completeUnion operation (Figure 31) starts by calling completeUnionGiver, which operates on the giving heap. This records the current heap indirection of the receiver. If the heap's head is still labelled with the UnionGiver label (so another thread has not completed these steps), and if the heap indirection is the same as that recorded at the start of the union, it is updated to point to the receiver's heap indirection, and replaced by a new heap indirection.

Next, completeUnionOnGiver increments the epoch from the value stored in the UnionGiver label. It then updates the header node to remove the label and

set the next reference to null.

Finally, completeUnion calls clearUR to mark the UnionReceiver label as done, and remove it from the HeapLabelList.

Lemma 55 *The code in Figure 31 correctly implements steps 6–9 of the union.*

Proof: Recall (Rule 5) that a heap's heap indirection can change only when the heap has a UnionGiver label; hence the receiver's heap indirection will not change from the value read until after the union is completed; hence if the test at line 7 succeeds, the value in newHi is valid. Further, all threads that call setNext will pass in the same value; only the first thread will succeed; and for others, the assertion inside setNext will succeed. Similarly, only the first thread to replace the giving heap's heap indirection will succeed.

The remaining steps are straightforward. Again, only the first thread to attempt each step will succeed. □

The following lemma makes an assumption about nodes' heap indirections being initialised correctly, which we discharge in Lemmas 59 and 60.

Lemma 56 *Suppose that whenever a node is inserted into heap h, its heap indirection is set to h.heapIndirection; then the getHeap function on a node always returns the correct heap.*

Proof: It is an invariant that following the sequence of HeapIndirections from a node in h leads to h.heapIndirection, except while h is the giving heap in a union between steps 3 and 6: the assumption of the lemma establishes this; the code in completeUnionOnGiver re-establishes it for each union.

Hence, except between steps 3 and 6, the result of the lemma holds. At step 3, each node becomes a member of the receiving heap; at this point, the HeapIndirections still lead to the giving heap. Consider the call to heapOf on the final HeapIndirection in this chain in this case.

- If the second read of next gives the same result as the first, then step 6 has not happened, so the read of head in heapOf happened before step 8, and so heapOf correctly returns the receiving heap.
- If the second read of next gives a different result, then getHeap restarts: this is necessary in case the read of head in heapOf happened after step 8, and so returned that heap.

Thus heapOf correctly returns the receiving heap. □

11.6 Helping with a union

Figure 32 contains the code for helping with a union. (We explain the case of Inserted labels in Section 12.1.)

Helping with a UnionLast label is straightforward: it calls setDone, then clears the label, corresponding to the two transitions from state $(2, F, T, L)$ in Figure 29.

The subsidiary function helpUG helps with a UnionGiver label ug; this simply calls the appropriate sub-function: recall that all the earlier functions were written to allow helping threads, as long as the expected earlier steps have been performed. If the corresponding UnionReceiver label ur has been added, it calls either clearLast or unionJoin, depending on whether the joining CAS has been

```
1  private def help(helpNode: Node, helpState: NodeState) = helpState.label match{
2    ...
3    case UnionLast(ug) =>
4      ug.setDone(helpNode, helpState); clearLastLabel(helpNode, helpState)
5    case ug: UnionGiver => helpUG(helpNode, helpState, ug)
6    case hll: HeapLabelList => hll.close
7    case Inserted(node, ia) =>
8      assert(helpState.next == node); clearInserted(node, helpNode, helpState, ia)
9  }
10 /** Help with a UnionGiver label. */
11 private def helpUG(helpNode: Node, helpState: NodeState, ug: UnionGiver) = {
12   val UnionGiver(receiver, ur) = ug; val giver = ur.giver
13   if(ug.joined) receiver.clearLast(giver, ug)
14   else if(receiver.labelledWith(ur)) receiver.unionJoin(giver, ug.first, ug)
15   else if(ug.aborted) giver.replaceLabel(ug, ug.replacedBy)
16   else{
17     if(!ug.isInit) giver.initUG(ug, helpState.next)
18     receiver.unionLabelReceiver(giver, ug.first, ug, true)
19   }
20 }
21 /** Is the head node labelled with a HeapLabelList containing ur? */
22 private def labelledWith(ur: UnionReceiver) = head.getLabel.match{
23   case hll: HeapLabelList => hll.getUR eq ur; case _ => false
24 }
25 /** Help with a UnionReceiver label. */
26 private def helpUnionReceiver(ur: UnionReceiver) = {
27   val giver = ur.giver
28   giver.head.getLabel match{
29     case ug @ UnionGiver(receiver, ur1) if ur1 eq ur =>
30       if(ug.joined) receiver.clearLast(giver, ug) else receiver.unionJoin(giver, ug.first, ug)
31     case _ => ur.done = true
32 } }
```

Figure 32: Helping with a union.

performed. If ug has been aborted, it helps to replace it with the appropriate label. Otherwise, it makes sure that ug is initialised, and helps add ur.

Helping with a HeapLabelList amounts to calling the close operation on it: recall that this helps with all the operations it contains, and then removes that label.

The function helpUnionReceiver is called by a HeapLabelList, either when it is being closed, or if another thread is trying to add a different UnionReceiver. If the corresponding UnionGiver is still on the head of the giving heap, it calls clearLast or unionJoin, appropriately. Otherwise, all the steps of the union except step 9 have been performed, so it simply marks ur as done (the calling code will then remove it from the HeapLabelList).

11.7 Correctness

Proposition 57 *The union operation works correctly: the overall effect is to transfer the keys of the giving heap to the receiving heap.*

Proof: This follows from the previous lemmas, and considering the overall effect of the updates (see Figure 19). In particular, Lemma 54 shows that the heaps are joined appropriately, with the linearization point happening at the joining

CAS. Lemmas 52 and 53 shows that there is a one-one relationship between UnionGiver and UnionReceiver labels; further a union operation re-labels the giving heap only if its earlier attempt was aborted. Lemma 55 shows that the union is completed correctly, tidying up after itself (but maybe leaving a HeapLabelList on the receiving heap. Finally, calls to the various helping functions call other functions only after the relevant prerequisite steps have been performed. □

We now prove a liveness result. Recall from the Introduction that our approach is to prove that the operation completes under the assumption that no other thread makes any update to any node; here, we consider the addition of a label to a HeapLabelList to be an update on the corresponding node.

Lemma 58 *Consider a call to union such that, from some point on, no other thread makes an update to any node. Then union completes after a finite number of steps other, perhaps, than steps involved in helping other operations. Likewise, any attempt to help with a union completes after a finite number of steps other, perhaps, than recursively helping.*

Proof: The giver-labelling step might need to help with other operations corresponding to labels on the head node, possibly several if the head has a HeadLabelList, but only finitely many. It will then successfully add the UnionGiver label.

Likewise, the receiver-labelling step may need to help finitely many other operations. If, as the result of a blocking loop, its UnionGiver label is aborted, it will also have to help with the operation that replaced it; but this will happen at most once (since no more labels are added, by assumption).

For the joining CAS, it is possible that, after other threads stop updating nodes, that some "helping" threads call setLastInfo with stale values; however, there will be finitely many such, and each will be cleared by the call to updateLast. Eventually, setLastInfo will be called with the correct last node, and then the joining CAS will succeed.

Steps 4–9 are straightforward: the code contains no loop or recursion, so is necessarily finite.

The result about helping follows in the same way. □

12 Adapting operations to deal with concurrent unions

In this section we discuss how to adapt the code for other operations to deal with concurrent unions.

Each operation on the receiving heap is unaffected (other than maybe having to help with a union). The union just has the effect of adding several nodes to the end of the root list, and the implementations already deal with this.

Also, merge is unaffected. If a thread executes tidy on the giving heap in a union, it may end up merging two trees in the *receiving* heap: but this is completely harmless. Note that tidy never returns to the head node of its heap after it has started traversing: to do so would risk merging nodes that were added to the giving heap after the union with nodes in the receiving heap.

12.1 Insertion

Insertion is somewhat harder. If the heap is empty and the head is labelled with a HeapLabelList, we allow the new node to be inserted after the head (as is allowed by Rule 4). While traversing to find a place to insert, we ignore the other types of labels we have introduced to support unions (including the Inserted labels that we introduce below): none can appear on the last node of the heap, so there is no need to help.

Further, if while traversing the thread discovers that the current node has been transferred to another heap, because of a giving union, it restarts the traversal. However, it might be necessary to help to complete the union, to ensure lock-freedom, as follows:

```
if (allowConcurrentUnion && curr != head && curr.getHeap != this){
  val headState = head.getState
  headState.label match{
    case ug : UnionGiver => helpUG(head, headState, ug); case _ => {}
  }
  restart
}
```

The variable allowConcurrentUnion is true if giving unions may be performed concurrently to other operations.

Recall, from Section 11, that we need to prevent the critical step (the inserting CAS) from happening concurrently to a giving union, or else the insertion may incorrectly happen on the receiving heap. To this end, we guard the critical steps by placing a suitable HeapLabel (Figure 20) on the header node within a HeapLabelList, thereby preventing a concurrent giving union. We consider, in turn, the two different ways of inserting a node.

For the case of inserting the new node below a root node of degree 0, we replace lines 21–23 of Figure 12 by

```
if (insertBelow(myNode, curr, currState)) return
else myNode.state = myNode.state[parent ↦ null]
```

where the code for insertBelow is in Figure 33. When allowConcurrentUnion is false, insertBelow acts equivalently to as in Section 6. When it is true, the code starts by recording the heap's epoch. If then checks that parent is still in the current heap. Assuming so, it sets the node's heap indirection to be that of the heap.

If then attempts to add a suitable InsertBelow label in a HeapLabelList to the heap's head node, via the function labelHead. If (in the case of insertBelow) parent's state changes, or this heap is the giving heap in a union, then the attempt to insert would fail (similar considerations apply with other functions, below). Therefore insertBelow passes parent and its state, and the current epoch to labelHead, so that if it detects a change it can fail (and return false). Otherwise, if the head has a null label, it attempts to add a new HeapLabelList containing the HeapLabel. If the head has an existing HeapLabelList, it attempts to add the HeapLabel to it. In other cases (including a UnionGiver), it helps with the label.

If the labelling succeeds, insertBelow calls completeInsertBelow; other threads may help with this part. It first checks that the epoch is unchanged, which implies that parent is still in the current heap. It then checks that label has not been marked as done, as an optimisation. Finally, it tries to perform the inserting CAS. Whether or not these steps succeed, it marks label as done, and tidies the HeapLabelList. If the CAS did not succeed, it needs to test whether

```
1  private def insertBelow(myNode: Node, parent: Node, parentState: NodeState): Boolean =
2    if(allowConcurrentUnion){
3      val ep = epoch.get
4      if(parent.getHeap == this){
5        myNode.heapIndirection.set(heapIndirection.get)
6        val label = InsertBelow(myNode, parent, parentState, ep)
7        labelHead(label, parent, parentState, ep) &&
8          completeInsertBelow(myNode, parent, parentState, label)
9      }
10     else false
11   }
12   else insertBelow0(myNode, parent, parentState)
13 private def insertBelow0(myNode: Node, parent: Node, parentState: NodeState): Boolean =
14   parent.compareAndSet(parentState, parentState[degree ↦ 1, children ↦ List(myNode)])
15 private def labelHead(label: HeapLabel, n: Node, nState: NodeState, ep: Long): Boolean = {
16   while(true){
17     if(n.getState != nState || epoch.get != ep) return false
18     else{
19       val headState = head.getState
20       headState.label match{
21         case null => // add new HeapLabelList
22           val hll = new HeapLabelList(this, label)
23           if(head.compareAndSet(headState, headState[label ↦ hll])) return true
24         case hll: HeapLabelList => if(hll.add(label)) return true
25         case l => help(head, headState)
26 } } } }
27 private def completeInsertBelow(
28     myNode: Node, parent: Node, parentState: NodeState, label: InsertBelow): Boolean = {
29   val ok =
30     epoch.get == label.epoch && !label.done && insertBelow0(myNode, parent, parentState)
31   label.done = true; tidyHeapLabelList
32   if(ok) true
33   else{ // test if another thread did the CAS
34     val newParentChildren = parent.getState.children
35     newParentChildren.nonEmpty && newParentChildren.last == myNode
36 } }
```

Figure 33: insertBelow and related functions.

some other thread has performed it; this is the case if and only if myNode is now the last child of parent.

Lemma 59 *Inserting as in Figure 33 works correctly. In particular: myNode is inserted into the heap this on which insert was called; myNode's heap indirection is set correctly; and the primary thread receives the correct result.*

Proof: Note that the InsertBelow is placed on the head node before the inserting CAS, and remains there until after that CAS. Hence during this period, there cannot be a giving union on this heap. Further, parent was found to be in the expected heap at line 4; if this heap were a giving heap in a union before the addition of InsertBelow, then the epoch would have increased; hence the check of the epoch at line 30 ensures that parent is still in the heap.

Similarly, if the epoch is unchanged, then the heap's heapIndirection is unchanged from where it was read at line 5. Hence myNode's heap indirection is initialised correctly.

If the first thread to attempt the CAS succeeds, then all other threads detect

this at line 35; note that the property of being parent's last child is maintained by all other operations. Hence all threads return true.

Conversely, if the first thread to test the epoch finds it changed, then so do all subsequent threads. And if the first thread to attempt the CAS fails, then so do all subsequent threads. In these cases, the thread that called insertBelow returns false (a helping thread may return true, at line 35, if a *subsequent* attempt by the primary thread leads to myNode being inserted below parent; but this result is ignored). □

We now consider the case of inserting at the end of the root list. We replace lines 15–17 of Figure 12 by

```
if (insertAtEnd(myNode, curr, currState)) return else BackOff()
```

where the code for insertAtEnd is in Figure 34. We call insertAtEnd in the same way if the heap is empty and the head is labelled with a HeapLabelList.

If allowConcurrentUnion is false, the operation is equivalent to as in Figure 12. Otherwise, if last = head, there is no need to add a HeapLabel: if a concurrent giving union happens, the state of head changes so the subsequent CAS fails; however, it is necessary to set the node's heap indirection.

In the remaining case, things are slightly more complex than with insertion below a root node, because there is no convenient way of detecting whether the insertion has been done by another thread. Our approach is to add an Inserted label (Figure 34) onto last at the same point as the inserting CAS. We then set the success flag in the InsertAtEnd HeapLabel ia, before removing the Inserted label. The relevant invariant is: the inserting CAS for myNode succeeded if and only if last has an Inserted label containing myNode or ia.success is true.

insertAtEnd first records the current epoch. It then checks that last is still in the current heap. If so, it sets myNode's heap indirection. It then calls labelHead to try to add an InsertAtEnd HeapLabel to the head, as long as last's state and the epoch do not change. If this is successful, it calls completeInsertAtEnd; other threads can help with this part.

completeInsertAtEnd checks that the epoch has not changed, and (as an optimisation) that another thread has not marked the InsertAtEnd label as done. In this case, it attempts to CAS the new node after last, adding an Inserted label at the same time. If this succeeds, it calls clearInserted to complete the update, and returns true. If either of the previous steps fails, it calls testIfInserted, to test if another thread has completed the insertion.

clearInserted sets the success flag in the InsertAtEnd label, to indicate to other threads that the inserting CAS succeeded. It then removes the Inserted label from last. Finally, it marks the InsertAtEnd label as done. Note (Figure 32) that another thread that encounters the Inserted label can help by calling clearInserted.

testIfInserted starts by testing if ia.done is set; if so, ia.success indicates whether the CAS was successful. Otherwise, if last has an Inserted label containing myNode, another thread did the CAS; this thread calls clearInserted to complete the update and returns true. Otherwise, it sets the done field in ia and returns the result of ia.success.

Lemma 60 *Inserting as in Figure 34 works correctly. In particular: myNode is inserted into the heap this on which insert was called; myNode's heap indirection is set correctly; and the primary thread receives the correct result.*

```
1  private def insertAtEnd(myNode: Node, last: Node, lastState: NodeState): Boolean = {
2    assert(lastState.next == null && lastState.parent == null &&
3      (lastState.label == null || last == head && lastState.label.isInstanceOf[HeapLabelList]))
4    if(allowConcurrentUnion && last != head){
5      val ep = epoch.get
6      if(last.getHeap == this){
7        myNode.heapIndirection.set(heapIndirection.get)
8        val label = InsertAtEnd(myNode, last, lastState, ep)
9        labelHead(label, last, lastState, ep) &&
10         completeInsertAtEnd(myNode, last, lastState, label)
11     }
12     else false
13   }
14   else{
15     if(allowConcurrentUnion) myNode.heapIndirection.set(heapIndirection.get)
16     last.compareAndSet(lastState, lastState.setNext(myNode))
17 } }
18 private def completeInsertAtEnd(
19     myNode: Node, last: Node, lastState: NodeState, ia: InsertAtEnd): Boolean =
20   if(epoch.get == label.epoch && !ia.done){
21     val lastStateL = lastState[next ↦ myNode, label ↦ Inserted(myNode, ia)]
22     if(last.compareAndSet(lastState, lastStateL)){
23       clearInserted(myNode, last, lastStateL, ia); true
24     }
25     else testIfInserted(myNode, last, ia)
26   }
27   else testIfInserted(myNode, last, ia)
28 private def clearInserted(myNode: Node, last: Node, lastStateL: NodeState, ia: InsertAtEnd) = {
29   ia.success = true; last.compareAndSet(lastStateL, lastStateL[label ↦ null])
30   ia.done = true; tidyHeapLabelList
31 }
32 private def testIfInserted(myNode: Node, last: Node, ia: InsertAtEnd): Boolean = {
33   if(ia.done) ia.success
34   else{
35     val lastState = last.getState
36     lastState.label match{
37       case l @ Inserted(n,_) if n == myNode => clearInserted(myNode, last, lastState, ia); true
38       case _ => ia.done = true; ia.success
39 } } }
40 /** node has just been inserted after the node of this label. */
41 case class Inserted(node: Node, ia: InsertAtEnd) extends Label
```

Figure 34: insertAtEnd and related functions, and the Inserted class.

Proof: Most parts of the proof are very similar to that for Lemma 59, so we omit them. To see that testIfInserted gives the correct result, note that the following invariant holds: the inserting CAS for myNode succeeded if and only if last has an Inserted label containing myNode or ia.success is true; and the Inserted label is removed before ia.done is set to true. □

12.2 Minimum

Recall that the minimum operation needs to guard against a concurrent giving union: less careful approaches risk returning a value from the receiving heap. However, if minimum detects that the joining CAS of a giving union has happened, it can immediately return None, with the operation linearized at some point between the joining CAS and the removal of the UnionGiver label, during

```
private def getEpoch: Option[Long] = {
  val ep = epoch.get
  head.getLabel match{
    case ug: UnionGiver if ug.joined => None
    case _ => if(epoch.get == ep) Some(ep) else None
} }
private def epochValid(ep: Long): Boolean = head.getLabel match{
  case ug: UnionGiver if ug.joined => false; case _ => epoch.get == ep
}
```

Figure 35: The getEpoch and epochValid funcitons.

which time the heap was empty. The revisions to the version of the minimum function in Section 8 are small, so we just explain the changes.

The revised minimum function starts by obtaining the current epoch using the function getEpoch (Figure 35). If there is a current giving union that has been linearized, then getEpoch returns None, and minimum also returns None. If getEpoch returns a value Some(startEpoch) then (as an optimisation) each time the traversal restarts, it tests whether the epoch has changed, and if so again returns None.

The advance function (Figure 10) is adapted so that from the head node of a giving heap in a union where the joining CAS has occurred, it returns a result indicating that the heap is empty:

```
case ug: UnionGiver =>
  result = skipDeleted(currState.next); if(ug.joined) result = (null, −1, List())
```

If minimum reaches the end of the traversal, and finds a candidate value to return, it again checks whether the heap has been the giving heap in a union. We replace line 20 of Figure 15 by

```
else if(! minNode.deleted){
  if(allowConcurrentUnion && !epochValid(startEpoch)) return None
  else return Some(minNode.key)
}
```

The function epochValid checks that its argument is still the valid epoch, taking into account a union that has been linearized but for which the epoch variable hasn't been incremented.

There is another subtlety. The traversal might encounter a node n that was marked for deletion while it was in some other heap h, but h has subsequently been the giving heap in a union with the current heap; in this case, it would be incorrect to return n's key. We deal with this by extending Delete labels to include a heap field, recording the heap of the deletion. We then replace line 8 of Figure 15 by the following, to ignore nodes deleted in another heap.

```
case delLabel: Delete
  if delLabel.ts − startTime <= 0 || allowConcurrentUnion && delLabel.heap != this =>
    insertNodes(nState.children)
```

Proposition 61 *The revised version of minimum works correctly, as stated in Proposition 44 and Lemma 45. Further, if minimum returns a proper result, there was no joining CAS of a giving union during the traversal.*

Proof: Most of the proof is as earlier, so we just sketch the changes. If the function detects that the heap has been the giving heap in a union, and returns None, we linearize the operation at some point between the joining CAS and when the UnionGiver label is removed: the heap contains no nodes at that point.

Suppose the function returns a proper result. Then necessarily no joining CAS of a giving union was concurrent to the traversal. Lemma 43 still holds; in particular when this heap is the receiving heap in a union, the new nodes are added *later* in the root list than node r. The proof in Proposition 44 then goes through as before: in particular, curr remains a root node of the current heap throughout the traversal.

Finally, we explicitly avoid adding a node to minList that had been marked for deletion in a different heap. When we consider deletion (Section 12.3) we will ensure that when the marking happens, the node really is in the appropriate heap, i.e. before the joining CAS of a giving union. Further, the starting epoch is recorded *before* startTime; hence, if, the node were marked in the current heap and transferred to another heap before the epoch was read, then transferred back to this heap, it would have been marked before startTime and so rejected. Hence, if the value returned is marked for deletion, that marking must have happened in the current heap during the current epoch. □

12.3 deleteMin

The traversal for deleteMin is revised in a similar way to that for minimum. At the start, the epoch is obtained using getEpoch: if this indicates that a giving union is happening, deleteMin can immediately return None. Otherwise, each time the traversal restarts, or when it finds a candidate node for deletion, if it finds that the epoch has increased, it again returns None.

Recall that we need to prevent a giving union concurrent to the labelling of the node delNode to be deleted. Our approach is similar to that for insertion: we add a HeapLabel (specifically a LabelForDelete) to the head node during the labelling of delNode. To this end, we replace line 3 of Figure 7 by

```
val delStateL =
  if(allowConcurrentUnion)
    labelHeadForDelete(pred, delNode, delState, delState.next, startEpoch)
  else labelForDelete(pred, delNode, delState, delState.next)
```

and similarly with line 16.

The function labelHeadForDelete (Figure 36) prepares a Delete label delLabel for delNode; recall (Section 12.2) we have extended such labels with a heap field to indicate the heap of the deletion. It then creates a LabelForDelete label lfd, and tries to add it to the head node using labelHead. If successful, it calls completeLabelForDelete; other threads may help with this.

completeLabelForDelete first checks that the epoch is not changed. If so, it tries to add the Delete label to delNode (similarly to as with labelForDelete). Whether or not, this succeeds, it marks lfd as done. If its own CAS failed, it checks whether another thread succeeded: this is the case if and only if delNode now contains the Delete label.

Further, the function findPred (Figure 11) needs to cope with the possibility of delNode having been moved to another heap by a giving union; and if it fails to find delNode, it again needs to consider the possibility that it has been moved to another heap. To that end, findPred is revised as in Figure 37. This starts by

```
1  private def labelHeadForDelete(pred: Node, delNode: Node, delState: NodeState,
2      newNext: Node, startEpoch: Long): NodeState = {
3    val delLabel = Delete(pred); delLabel.heap = this
4    val lfd = LabelForDelete(delNode, delState, newNext, startEpoch, delLabel)
5    if(labelHead(lfd, delNode, delState, startEpoch)
6      completeLabelForDelete(delNode, delState, newNext, startEpoch, lfd)
7    else null
8  }
9  private def completeLabelForDelete(delNode: Node, delState: NodeState, newNext: Node,
10     startEpoch: Long, lfd: LabelForDelete): NodeState = {
11   var delStateL: NodeState = null; var done = false
12   if(epochValid(startEpoch)){
13     delStateL = delState[next ↦ newNext, seq ↦ delState.seq+1, label ↦ lfd.delLabel]
14     done = delNode.compareAndSet(delState, delStateL)
15   }
16   lfd.done = true; tidyHeapLabelList
17   if(done) delStateL
18   else{ delStateL = delNode.getState; if(delStateL.label eq lfd.delLabel) delStateL else null }
19 }
```

Figure 36: The functions labelHeadForDelete and completeLabelForDelete.

```
1  private def findPred(delNode: Node): (Node, NodeState) = {
2    val ep = epoch.get; val h = delNode.getHeap
3    if(h != this) h.findPred(delNode, sentinel)
4    else{
5      ... // search as before
6      if(allowConcurrentUnion && epoch.get != ep && !delNode.deleted)
7        delNode.getHeap.findPred(delNode, sentinel)
8      else{ delNode.deleted = true; (delNode, null) }
9  } }
10 private def completePredUpdate(
11     pred: Node, predState: NodeState, delNode: Node, next: Node) : Boolean = {
12   assert(predState.next == delNode && predState.parentless)
13   predState.label match{
14     case null => completePredUpdate0(pred, predState, delNode, next, null)
15     case hll: HeapLabelList =>
16       val ur = hll.getUR
17       if(ur != null) ur.giver.head.getLabel match{
18         case ug @ UnionGiver(_, ur1) if ur eq ur1 => ug.joined
19         case _ => {} // The UnionGiver has been removed, so already marked as joined
20       }
21       completePredUpdate0(pred, predState, delNode, next, hll)
22     case _ => help(pred, predState); false
23 } }
```

Figure 37: Revised versions of findPred and completePredUpdate; the function completePredUpdate0 encapsulates the code in the null case of Figure 8.

storing, in ep, the current epoch. It then calls getHeap on delNode; if it is now in a different heap h, it recurses on that heap. Otherwise, it searches as in Figure 11. If this fails to find delNode, but the epoch has changed, and the deleted flag has not been set, it needs to search again in the new heap. Otherwise, delNode really has been decoupled, so its deleted flag can be set.

Finally, we allow completePredUpdate to be performed when the predecessor node has a HeapLabelList. However, we need to do so following Rule 4. Figure 37

```
private def helpHL(label: HeapLabel) = label match{
  case ib @ InsertBelow(node, parent, parentState, epoch) =>
    completeInsertBelow(node, parent, parentState, ib)
  case ia @ InsertAtEnd(node, pred, predState, epoch) =>
    completeInsertAtEnd(node, pred, predState, ia)
  case lfd @ LabelForDelete(delNode, delState, newNext, startEpoch, delLabel) =>
    completeLabelForDelete(delNode, delState, newNext, startEpoch, lfd)
}
```

Figure 38: The helpHL function.

does this: if a receiving union is underway, it calls joined on the corresponding UnionGiver to ensure that if the joining CAS has happened, this is recorded.

Lemma 62 *The revised version of findPred returns a correct result, as in Lemma 25, except without the premiss of no concurrent giving unions.*

Proof: The proof is mostly as earlier. Consider the case (line 8 of Figure 37) that findPred fails to find delNode, deduces that it has been decoupled by another node and so marks it as deleted. delNode was in the current heap at line 2. Since then, the epoch hasn't changed, so step 8 of a giving union on this heap has not happened since (if the epoch was read between steps 7 and 8 of a union, then the search will be in the *receiving* heap). Hence, if delNode had not been decoupled, it would have been reachable from head throughout this period, and so the search would have found it. □

Proposition 63 *The correctness results for deleteMin (Propositions 15 and 47) still hold. Further, if deleteMin returns a proper result, there was no joining CAS of a giving union between the start of the traversal and the marking of the node for deletion.*

Proof: The proof is again mostly the same as the proofs of earlier results, so we simply sketch the differences. The case of returning None because a concurrent giving union is detected is the same as in Proposition 61.

The code in Figure 36 ensures that no concurrent giving union happens between the start of the first traversal and when the node is labelled. Hence delNode really is in the expected heap when labelled. The code at the end of completeLabelForDelete correctly identifies whether another thread has labelled delNode (recall that Delete labels are never removed). □

12.4 Helping with HeapLabels

Figure 38 gives the function helpHL, which is called by a HeapLabelList to help to complete the operation corresponding to a HeapLabel. Each branch simply helps to complete the corresponding part of the operation.

12.5 Liveness

Lemma 64 *Suppose, from some point on, other threads perform no updates on nodes. Then each of the new versions of operations performs a finite number of steps, other, perhaps, than helping. Likewise, each attempt to help one of these*

operations performs a finite number of steps other, perhaps, than recursively helping.

13 Linearization

In this section we consider linearizability of the binomial heap. Note that for minimum and deleteMin operations, it is enough to consider just the *final* (successful) traversal. Below, we say just "traversal" to mean the final traversal. Note also that we can concentrate on a single heap: a history is linearizable if its restriction to each object is linearizable [HW90]; for both heaps, we linearize each union at the joining CAS.

We need to capture what are legal histories for a sequential binomial heap. We describe histories using events of the form $op_o(args)$: *res* to represent a call to operation *op* on heap object o, with arguments *args*, returning *res*; we omit *args* or *res* when null; we omit o when obvious from the context. A sequential history is a sequence of such events. The state of a binomial heap after a history h is the multiset of keys that have been added but not yet removed (we use set notation to describe multisets):[18]

$$\begin{aligned} stateAfter_o(h) = \{k \mid \mathsf{insert}_o(k) \in h\} \cup \\ \bigcup\{stateAfter_{o'}(h') \mid h' +\!\!+ \langle \mathsf{union}_o(o')\rangle \leq h\} - \\ \{k \mid \mathsf{deleteMin}_o\colon \mathsf{Some}(k) \in h\}. \end{aligned}$$

Definition 65 *A sequential history h is legal for a binomial heap if*

- *For each prefix $h' +\!\!+ \langle minimum\colon None\rangle$ or $h' +\!\!+ \langle deleteMin\colon None\rangle$ of h, the heap is empty after h'; i.e. $stateAfter(h') = \{\}$;*
- *For every prefix $h' +\!\!+ \langle minimum\colon Some(k)\rangle$ or $h' +\!\!+ \langle deleteMin\colon Some(k)\rangle$ of h, k is the minimum element of the heap after h': $k = \min(stateAfter(h'))$.*

The following lemmas will be useful later.

Lemma 66 *Suppose:*

1. *$h_1 +\!\!+ \langle deleteMin\colon Some(k)\rangle +\!\!+ h_2 +\!\!+ h_3$ is a legal sequential history;*
2. *k is the smallest key in the heap after $h_1 +\!\!+ h_2$;*
3. *h_2 contains no $deleteMin\colon Some(k')$ or $minimum\colon Some(k')$ operation with $k' > k$, no $deleteMin\colon None$ or $minimum\colon None$ operation, and no giving union.*

Then $h_1 +\!\!+ h_2 +\!\!+ \langle deleteMin\colon Some(k)\rangle +\!\!+ h_3$ is a legal sequential history.

Proof: Consider just operations on ho. We need to show that, within $h_1 +\!\!+ h_2 +\!\!+ \langle \mathsf{deleteMin}\colon \mathsf{Some}(k)\rangle +\!\!+ h_3$, each key returned by a deleteMin or minimum operation is indeed the minimum key at that point, and that if such an operation returns None, the heap is empty at that point. This is true of the $\mathsf{deleteMin}\colon \mathsf{Some}(k)$ operation, by assumption 2. It is true of any such operation within h_1 or h_3, since the state of the heap is the same as at the corresponding point in the

[18] We use $\leq$ to denote history prefixing.

original history. By assumption 3, no deleteMin or minimum operation within h_2 returns None.

Consider a prefix

$$h_1 +\!\!+ h_2' +\!\!+ \langle \mathsf{deleteMin}\colon \mathsf{Some}(k')\rangle \qquad \text{or} \qquad h_1 +\!\!+ h_2' +\!\!+ \langle \mathsf{minimum}\colon \mathsf{Some}(k')\rangle$$

of $h_1 +\!\!+ h_2$. By assumption 3, $k' \leq k$. Now, k' was the minimum key after $h_1 +\!\!+ \langle \mathsf{deleteMin}\colon \mathsf{Some}(k)\rangle +\!\!+ h_2'$, so it must also be the minimum key after $h_1 +\!\!+ h_2'$. □

Lemma 67 *Consider two concurrent minimum or deleteMin operations op_1 and op_2 of keys k_1 and k_2 with $k_1 < k_2$, on the same heap, such that op_1 finishes its traversal before op_2. Then k_1 must have been marked (and maybe removed from the heap) before op_2 finished its traversal.*

Proof:

If k_1 had not been marked, then op_2 would have returned k_1, by Propositions 44 and 47. □

Below we talk about various events: the invocation and returns of operations; the successful CAS for an insertion; the end of the final traversal for a minimum or deleteMin; the marking of a node for deletion; the completion of the deletion; or a linearization point. As is standard, we assume that no two events occur at the same time, except we sometimes define a linearization point to be at the same time as another event of the same operation: if two other events, necessarily by different threads, really do happen at the same time, then they could have happened in either order. Below we sometimes define a time t to be "immediately after" some event: by this, we mean that t happens after that event but before any other.

Theorem 68 *The binomial heap implementation is linearizable.*

Proof: We describe, given an execution, how to build a corresponding legal sequential history suitable for linearization. Consider an intermediate state within an execution. We define the corresponding abstract state for each heap, as a multiset of keys, as follows: a key k is in the multiset if it is in the heap (so the joining CAS of a giving union has not transferred it to another heap), and the instance of deleteMin that deletes the corresponding node (if any) has not yet finished its traversal. Note that this is a subset of the unmarked keys in the heap.

We start by linearizing the insert, union, deleteMin and unsuccessful minimum operations in a way that is sound with respect to this abstraction; later we explain how to add successful minimum operations, which may require moving the linearization points of deleteMin operations, and re-defining the abstract state.

- We linearize each insert or union at the point of its successful inserting or joining CAS. This corresponds to the abstract state defined above.
- We linearize each successful deleteMin at the end of its (final) traversal. Suppose it returns key k. Propositions 47 and 63 tell us that the node in question was a minimal unmarked node at this point; hence, since the abstract state is a subset of the unmarked keys, k was the smallest key in the corresponding abstract state, as required.

- We linearize each unsuccessful minimum or deleteMin at the point, implied by Propositions 44 and 61, or 47 and 63, at which the heap has no unmarked keys; so again the linearization point corresponds to the abstract state.

We now seek to linearize the successful minimum operations. In most cases, we linearize the operation when the traversal finishes. However, if a concurrent deleteMin traversal has already identified the node in question as the minimum (but, necessarily, not yet completed the deletion), then we need to find a different, earlier, linearization point for the minimum; this may also require moving the linearization point of the deleteMin. We re-interpret the corresponding abstract state so that elements are removed at that new linearization point.

We pick linearization points for successful minimum operations in increasing order of keys. Suppose we have dealt with all keys $k' < k$, and we are looking to linearise minimum operations for key k. Our induction hypothesis is that we have found linearization points for all insertions, unions, deleteMin operations, and unsuccessful minimum operations, and all minimum operations for keys $k' < k$, and maybe some for key k (but none with larger keys), and we have defined a corresponding abstract state, such that:

IH1. Each insert or union operation is linearized when its keys are added to the abstract state;

IH2. Each unsuccessful deleteMin or minimum operation is linearized at a point at which there was no unmarked key in the heap, and the corresponding abstract state was empty;

IH3. Each deleteMin operation giving a key $k' \leq k$, and each minimum operation that has been linearized so far (also giving a key $k' \leq k$) is linearized at a point at which k' was the smallest key in the abstract state;

IH4. Each deleteMin operation giving a key $k' > k$ is linearized at the end of the traversal, at which point k' is the smallest key in the abstract state;

IH5. For each minimum operation not linearized so far (so giving a key $k' \geq k$), at the end of its traversal there is no key $k'' < k$ in the abstract state.

The above construction establishes the inductive hypothesis for $k = -\infty$, using Propositions 44 and 61 for IH5.

Consider a particular node n with key k. Consider a minimum operation that returns the key from n. Let t be the time that the corresponding deleteMin finished its traverse (including, possibly, in a different heap); or let $t = \infty$ if there is no such deleteMin.

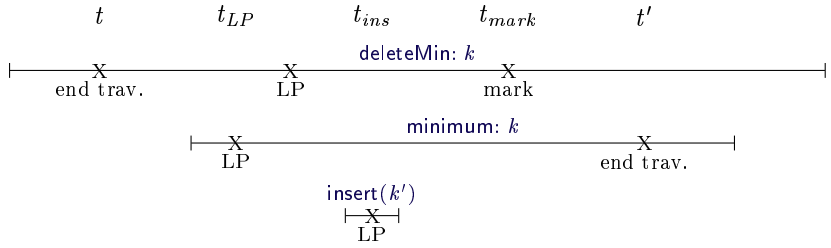
Case 1. If such a minimum operation for node n completed its traverse at time $t' < t$, we can linearize that minimum at t'. The inductive hypothesis (IH5) tells us that there was no smaller key in the abstract state at that point; and the deleting operation (if any) had not yet finished its traversal, so k is still in the abstract state.

Now consider those calls to minimum (that return the key from n) that complete their traverse at some time $t' > t$. Let $t_{mark} > t$ be the time that n is marked for deletion. Note that each such minimum started before t_{mark}, by Lemma 45; but t' might be either before or after t_{mark}. Let t_{ins} be the time of the linearization point of the first insertion[19] after t of a key $k' < k$, or ∞

[19]For the rest of this proof, when we talk about insertion of keys, we intend it to include receiving union operations.

if there is no such insertion (if $t_{ins} < t'$, then that key must have been deleted before t').

Case 2. If the start point of a minimum operation for node n is before t_{ins}, then we linearize it at some time t_{LP} after it started and before each of t_{mark}, t' and t_{ins}. We will linearize the deleteMin at a time later than t_{LP}, and remove k from the abstract state at that time: see below. This is illustrated in the figure below (where "LP" indicates the linearization points, and t_{ins}, t_{mark} and t' may occur in any order).



By the inductive hypothesis (IH3), k was the smallest key in the abstract state at t (since the deleteMin giving k was previously linearised at t). No smaller key is inserted before t_{LP}; and k is not removed before t_{LP}. There is no giving union between t and t_{mark} by Proposition 63. Hence k is still the smallest key in the (updated) abstract state at t_{LP} (re-establishing IH3 for this minimum).

We now consider the linearization point for deleteMin.

a. If the start point of *every* such minimum is before t_{ins}, then we linearize the deleteMin immediately after the last linearization point for such a minimum (as illustrated above). As above, k is the smallest key in the abstract state at this point (re-establishing IH3 for this deleteMin). We need to show that moving the linearization point of the deleteMin cannot invalidate the linearization point of any other deleteMin or minimum operation. By Lemma 66, it is enough to show that the deleteMin has not been moved past any deleteMin or minimum that is unsuccessful or returns a key $k'' > k$.

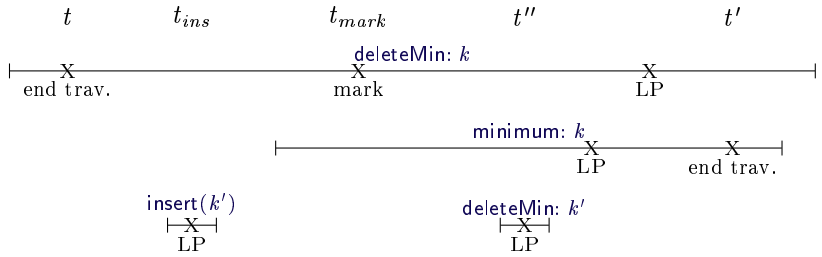
   - The key k is in the heap, unmarked, throughout this period, so (by IH2) there can be no such unsuccessful operation.
   - By the induction hypothesis, there is not yet any linearization point for a minimum operation for a key $k'' > k$ in the sequential history.
   - By IH4, any deleteMin for key $k'' > k$ is currently linearized when it finishes its traversal; and by Lemma 67, part 1, no such deleteMin operation can have completed its traversal between t and t_{mark}.

   Hence moving the linearization point of the deleteMin in this way preserves the legality of the sequential history.

   We also need to prove that moving the linearization point does not falsify IH5 for any minimum operation for a key $k'' > k$. However, no such minimum can complete its traversal between t and the new linearization point for the deleteMin: if it had, it would have seen k unmarked, and so returned k rather than k''.

b. If some such minimum starts after t_{ins}, we linearize the deleteMin as in case 3, below.

Case 3. The final case arises if a key $k' < k$ is inserted with linearization point at some time t_{ins} after t, and a minimum (on node n) starts after t_{ins}; necessarily $t_{ins} < t_{mark}$ since the minimum starts before t_{mark} (Lemma 45). See figure below (again t' might be either before or after t_{mark}).



No giving union can occur between t and t_{mark} by Proposition 63, nor between the start of the traverse in minimum and t' by Proposition 61; combining these means no giving union can occur between t and $\max(t', t_{mark})$.

Note that, by IH5, k' must have been removed from the abstract state before t', as must every other key less than k that was inserted since t. Let $t'' < t'$ be the time of the last linearization point for such a deletion (t'' might be either before or after t_{mark}). So, immediately after t'', the abstract state holds no key less than k (by IH4). We linearize each such minimum immediately after t'' (so before t') in some order; and linearize the deleteMin immediately after the last such; this re-establishes IH3 for both operations.

The linearization point for the deleteMin is within its operation: the deleteMin is linearized before the last t', which is before the node n is fully deleted, which is before the end of the deleteMin operation, as required.

We need to show that moving the linearization point of the deleteMin does not invalidate the linearization point of any other operation. As in the previous case, it is enough to show that the deleteMin has not moved past any other deleteMin operation that is unsuccessful or returns a key $k'' > k$, or past an unsuccessful minimum operation. By IH2 and IH4, each such operation is linearized at the end of its traversal. But no such operation can finish its traversal between t and t'': between t and t_{mark}, that traversal would have seen k unmarked, and so returned k instead of failing or returning k''; and between t_{ins} and t'', there was a smaller unmarked key (e.g. k') in the heap. Hence shifting the linearization point preserves the legality of the sequential history.

We also need to show that moving the linearization point does not falsify IH5 for any minimum operation for a key $k'' > k$. However, no such minimum can complete its traverse between t and the new linearization point for the deleteMin of k: as in the previous paragraph, it would have seen a smaller unmarked key, and so returned that instead of k''. □

14 Lock-freedom

In this section we prove lock-freedom. We need to consider chains of helping, where a thread helping with one operation is forced to help another operation.

Lemma 69 *Suppose, from some point on, other threads make no update to nodes. Then each chain of helping is finite (so acyclic).*

Proof: The only types of labels for which helping may lead to recursively helping with another label are Delete, UnionGiver and UnionReceiver (within a HeapLabelList).

We show that any chain of helping traverses a finite sequence of heaps. Helping a UnionGiver label can require helping the label on the head of the receiving heap; if the latter is also a UnionGiver label, this can lead to recursive helping of its receiving heap; however, the code in breakLoop is designed to detect and break what might otherwise be a loop of unions helping one another. In addition, helping a UnionGiver label can lead to helping the label on the last node of the receiving heap; but the chain does not progress to a third heap from there.

Now consider a chain of Deletes helping one another. Combining Lemmas 16 and 26, each such instance of helping is via one of the following possibilities:

- from predUpdate, a call to findPred, leading to a call to helpDelete on the parent of delNode's predecessor;
- from deleteWithParent, a call to helpDelete on delNode's parent;
- from completePredUpdate, a call to help on delNode's predecessor.

In each case the node being helped is either closer to the root list or earlier in the root list than delNode, or has already been decoupled (in which case it's the final node in the chain of helping); thus this chain of Deletes must be finite.

If the chain reaches a node with a UnionGiver label, necessarily via the third case above, threads help with that. However, as above, helping never leads back to a node in this heap, so this does not lead to a cycle of helping.

If the chain reaches a HeapLabelList then the first pred update is done: it is important that the deletes do *not* help with a UnionReceiver in this case, for this could lead to a loop of helping (specifically if the union is attempting the joining CAS, and all the nodes of the receiving heap have a Delete label). □

Theorem 70 *The concurrent binomial heap is lock-free.*

Proof: Suppose, for a contradiction, that there is an infinite execution of the binomial heap, during which only a finite number of public operations (i.e. insert, minimum, deleteMin or union) complete. This assumption means that only a finite number of new calls are made to tidy (since tidy is called only when an insert, deleteMin or union finishes).

We show that, from some time onwards, no updates are made to any nodes.

1. Each public operation makes a bounded number of updates to nodes *excluding* (for the moment) the addition of HeapLabels within labelHead: for example, a union makes at most five such updates. Hence, from some time t_0 onwards, no such update is made to any node.
2. Consider merge operations. From time t_0 on, only a finite number of b updates of merges can take place (bounded by the number of roots, minus one). Hence, from some point on, no changes are made to the root list. Thus, by Lemma 51, from some time onwards, no thread performs any step within tidy (or its subfunction merge).

Operation probabilities			Initial	This paper	Sundell &	Lindén &
insert	deleteMin	minimum	size		Tsigas	Jonsson
0.5	0.5	0.0	10K	4012K±47K	2904K±40K	1386K±137K
0.5	0.5	0.0	100K	3270K±29K	2912K±37K	1797K±78K
0.5	0.5	0.0	1000K	1574K±42K	2499K±42K	1854K±195K
0.4	0.4	0.2	100K	3531K±46K	3581K±33K	2859K±234K
0.3	0.3	0.4	100K	3827K±43K	4475K±232K	4233K±624K
0.6	0.4	0.0	0	3525K±67K	2144K±277K	1953K±70K
0.7	0.3	0.0	0	2289K±97K	1569K±140K	1690K±67K

Operation probabilities				Initial	Number	This paper
insert	deleteMin	minimum	union	size	of heaps	
0.38	0.38	0.19	0.05	100K	4	1352K±10K
0.38	0.38	0.19	0.05	100K	16	2773K±531K
0.4	0.4	0.19	0.01	100K	16	3868K±39K

Figure 39: Experimental results, giving throughput (in operations per second).

3. Finally, each operation considered in step 1 needs to add a HeapLabel more than once only if some other update has interfered with the first attempt; so from some time t onwards each operation adds at most a single HeapLabel. Hence from some time, no more such additions are made, i.e. *no* updates are made to any nodes by these operations.

Then, by Lemmas 38, 40, 46, 49, 58 and 64, each call to insert, union, minimum and deleteMin returns after a finite number of steps, outside of helping; and each instance of helping takes a finite number of steps, outside recursive helping. Lemma 69 shows that each chain of helping is finite. Finally, each attempt at helping makes progress, so each operation helps with each node a finite number of times; and, by assumption, no new nodes are added, so each thread helps a finite number of nodes. Hence each operation performs a finite number of steps in total. □

15 Conclusions

In this paper we have presented a lock-free linearizable concurrent binomial heap. Our main interest was in the development and correctness of the implementation, rather than out-and-out performance. Nevertheless, the heap gives good performance in several situations. Figure 39 gives experimental results. The experiments were run on a 32-core server (two 2.1GHz Intel(R) Xeon(R) E5-2683 CPUs with hyperthreading enabled, with 256GB of RAM). In each execution, 64 threads performed two million randomly chosen operations each. Each row of each table corresponds to a particular choice of parameters, namely the probabilities of different operations, and the initial number of keys in the priority queue. In each case, ten executions were made, and the throughput calculated; the tables give the means and 95% confidence intervals.

The first table compares against the skiplist-based priority queues of Sundell and Tsigas [ST05], and Lindén and Jonsson [LJ13]. Our implementation gives the best throughput in most cases. It performs less well when it contains a large number of keys: the longer root list has an adverse effect on performance;

the corresponding effect on the skiplist-based implementations is smaller. The other implementations are more efficient on the minimum operation, since this simply selects the first key in the skiplist.

The second table includes the union operation (which is not provided by the other implementations). Throughput is still decent with reasonably infrequent union operations, and with a reasonable number of heaps. However, unsurprisingly, throughput is reduced with a small number of heaps or with frequent union operations: the union operation blocks the critical steps of other operations; and the HeapLabelList on the head node becomes a bottleneck.

We review some of the main challenges, and our solutions, in the hope that the solutions can prove useful in other applications. One of the biggest challenges was the problem of reliably traversing the heap, for example to find the smallest node, while other threads are rearranging it: our use of sequence numbers effectively allows one thread to signal to another that its traversal might have been corrupted. In addition, avoiding race conditions proved challenging: our use of labels to lock nodes helped avoid most of these.

Allowing giving unions concurrently to other operations presented a particular challenge. Our labelling of head nodes prevented such unions concurrently to the critical steps of those operations (while maintaining lock-freedom); but in addition, it was necessary to develop techniques to test whether a union had happened during some period, and to find the current heap containing a node; and allowing threads to help with the operation introduced additional difficulties.

There were several subtleties concerning linearizability; in particular, the fact that minimum can return a key that is marked for deletion before the traversal ends complicated the proof of linearizability. As noted in the Introduction, the techniques from [Low17] proved very useful in detecting subtle linearizability bugs in early versions, and strengthens our confidence in the final version.

Ensuring lock-freedom also proved challenging. In earlier versions of our implementation, we ran into a number of situations where a thread could get into a loop: for example, an attempted traversal could follow a loop of nodes because of an incomplete merge; or a chain of helping of operations that forms a loop. This difficulty is reflected in the complexity of the rank used to prove insert lock-free (Lemma 38).

We also encountered various challenges in trying to make the operations efficient. Allowing a thread to insert below an existing singleton node, rather than just at the end of the root list, removed a bottleneck. Preventing merges from using those nodes reduced interference. Our use of HeapLabelLists allowed the critical steps of operations to be concurrent to one another and to a receiving union, but not to giving unions. Nevertheless, when most insertions are of a key smaller than nearly all the keys in the heap, it becomes harder to insert below an existing node, thereby making the final root a bottleneck again; we conjecture this could be improved by using a combining funnel [SZ98] to allow several insertions to combine together into a single update on the heap.

For deleteMin, maintaining multiple minima while traversing made a huge difference: different deleteMin calls compete for the same nodes, so many fail; conversely, compiling a list of fall-back options does not cost much. Also, allowing a thread to mark a non-root node for deletion, when its parent is being deleted, gives benefits: it means that only the thread that succeeded in the marking has to help with the deletion of the parent.

Within delete, we previously labelled both the node to be deleted and its predecessor. However, this meant that *two* nodes were locked, rather than one; and it also meant that there was the possibility of the deletion being disrupted if the nodes were separated before the labelling, or the state of the second node changed after the first node was labelled. Locking just a single node seems to work better, even though it means that sometimes an additional traversal is necessary to find the new predecessor. This change also made achieving, and proving, lock-freedom easier: previously it was difficult to avoid cycles of operations helping one another. However, we can see no easy way to similarly reduce the amount of labelling in the merge operation.

Laziness can help. When finding the predecessor of the node being deleted, we use an inexpensive but unreliable search, before falling back on a reliable but expensive one. Likewise, within insert, we avoid helping an operation on the final root on the first few traversals, in the hope it will complete anyway. We suspect similar tactics can be used elsewhere.

Our implementation used various parameters, that can be tuned. It would be useful to make these *adaptive*, where they vary automatically. For example, the number of minimal nodes that a thread records while traversing in minimum or deleteMin could vary based on how many such nodes were actually used in previous operations; and the frequency of tidying could vary based on the length of the root list that a thread encounters while traversing.

References

[AKLS15] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: A scalable relaxed priority queue. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP '15)*, 2015.

[Bar93] Greg Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[BCP16] Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. CBPQ: High performance lock-free priority queue. In *Proceedings of Euro-Par 2016: Parallel Processing*, 2016.

[CDP96] Vincenzo A. Crupi, Sajal K. Das, and M. Cristina Pinotti. Parallel and distributed meldable priority queues based on binomial heaps. In *International Conference on Parallel Processing*, 1996.

[CLR99] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1999.

[GF64] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Communications of the ACM*, 7:301–330, 1964.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[HW90] Maurice Herlihy and Jeannette Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[HW91] Qin Huang and William E. Weihl. An evaluation of concurrent priority queue algorithms. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 518–525, 1991.

[LJ13] Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *OPODIS*, number 8304 in LNCS, pages 206–220, 2013.

[Low17] Gavin Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29(4), 2017.

[LS00] Itay Lotan and Nir Shavit. Skiplist-based concurrent priority queues. In *Proceedings of the First International Parallel and Distributed Processing Symposium*, 2000.

[LS12] Yujie Liu and Michael Spear. Mounds: Array-based concurrent priority queues. In *Proceedings of 41st International Conference on Parallel Processing*, 2012.

[MS96] Maged Michael and Michael Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[ST05] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

[SZ98] Nir Shavit and Asaph Zemach. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 1998.

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.