Pragmatic Primitives for Non-blocking Data Structures

Trevor Brown  
University of Toronto

Faith Ellen  
University of Toronto

Eric Ruppert  
York University

May 23, 2013

ABSTRACT
We define a new set of primitive operations that greatly simplify the implementation of non-blocking data structures in asynchronous shared-memory systems. The new operations operate on a set of Data-records, each of which contains multiple fields. The operations are generalizations of the well-known load-link (LL) and store-conditional (SC) operations, which we call LLX and SCX. The LLX operation takes a snapshot of one Data-record. An SCX operation by a process \( p \) succeeds only if no Data-record in a specified set has been changed since \( p \) last performed an LLX on it. If successful, the SCX atomically updates one specific field of a Data-record in the set and prevents any future changes to some specified subset of those Data-records. We provide an implementation of these new primitives from single-word compare-and-swap and prove it correct. As a simple example, we show how LLX and SCX can be used to implement a non-blocking multiset data structure in a straightforward way.

1. INTRODUCTION
Building a library of concurrent data structures is an essential way to simplify the difficult task of developing concurrent software. There are many lock-based data structures, but locks are not fault-tolerant and are susceptible to problems such as deadlock [11]. It is often preferable to use hardware synchronization primitives like compare-and-swap (CAS) instead of locks. However, the difficulty of this task has inhibited the development of non-blocking data structures. These are data structures which guarantee that some operation will eventually complete.

Our goal is to facilitate the implementation of high-performance, provably correct, non-blocking data structures on any system that supports a hardware CAS instruction. We introduce three new operations, load-link-extended (LLX), validate-extended (VLX) and store-conditional-extended (SCX), which are natural generalizations of the well known load-link (LL), validate (VL) and store-conditional (SC) operations. We provide a practical implementation of our new operations from CAS. Complete proofs of correctness appear in [7]. We also show how these operations make the implementation of non-blocking data structures and their proofs of correctness substantially less difficult, as compared to using LL, VL, SC, and CAS directly.

LLX, SCX and VLX operate on Data-records. Any number of types of Data-records can be defined, each type containing a fixed number of mutable fields (which can be updated), and a fixed number of immutable fields (which cannot). Each Data-record can represent a natural unit of a data structure, such as a node of a tree or a table entry. A successful LLX operation returns a snapshot of the mutable fields of one Data-record. (The immutable fields can be read directly, since they never change.) An SCX operation by a process \( p \) is used to atomically store a value in one mutable field of one Data-record and finalize a set of Data-records, meaning that those Data-records cannot undergo any further changes. The SCX succeeds only if a specified set of Data-records has not changed since \( p \) last performed LLX operations on them. A successful VLX on a set of Data-records simply assures the caller that each of these Data-records has not changed since the caller last performed an LLX on it. A more formal specification of the behaviour of these operations is given in Section 3.

Early on, researchers recognized that operations accessing multiple locations atomically make the design of non-blocking data structures much easier [5, 13, 17]. Our new primitives do this in three ways. First, they operate on Data-records, rather than individual words, to allow the data structure designer to think at a higher level of abstraction. Second, and more importantly, a VLX or SCX can depend upon multiple LLXs. Finally, the effect of an SCX can apply to multiple Data-records, modifying one and finalizing others.

The precise specification of our operations was chosen to balance ease of use and efficient implementability. They are more restricted than multi-word CAS [13], multi-word RMW [1], or transactional memory [17]. On the other hand, the ability to finalize Data-records makes SCX more general than \( k \)-compare-single-swap [15], which can only change one word. We found that atomically changing one pointer and finalizing a collection of Data-records provides just enough power to implement numerous pointer-based data structures in which operations replace a small portion of the data structure. To demonstrate the usefulness of our new operations, in Section 5, we give an implementation of a simple, linearizable, non-blocking multiset based on an ordered, singly-
linked list. Our implementation of an object that supports LLX, VLX, and SCX is designed for an asynchronous system where processes may crash. We assume shared memory locations can be accessed by single-word CAS, read and write instructions. We assume a safe garbage collector (as in the Java environment) that will not reallocate a memory location if any process can reach it by following pointers. This allows records to be reused.

Our implementation has some desirable performance properties. A VLX on $k$ Data-records only requires reading $k$ words of memory. If SCXs being performed concurrently depend on LLXs of disjoint sets of Data-records, they all succeed. If an SCX encounters no contention with any other SCX and finalizes $f$ Data-records, then a total of $k + 1$ CAS steps and $f + 1$ writes are used for it and the $k$ LLXs on which it depends. We also prove progress properties that suffice for building non-blocking data structures using LLX and SCX.

2. RELATED WORK

Transactional memory [12, 17] is a general approach to simplifying the design of concurrent algorithms by providing atomic access to multiple objects. It allows a block of code designated as a transaction to be executed atomically, with respect to other transactions. Our LLX/VLX/SCX primitives may be viewed as implementing a restricted kind of transaction, in which each transaction can perform any number of reads followed by a single write and then finalize any number of words. It is possible to implement general transactional memory in a non-blocking manner (e.g., [11, 17]). However, at present, implementations of transactional memory in software incur significant overhead, so there is still a need for more specialized techniques for designing shared data structures that combine ease of use and efficiency.

Most shared-memory systems provide CAS operations in hardware. However, LL and SC operations have often been seen as more convenient primitives for building algorithms. Anderson and Moir gave the first wait-free implementation of small LL/SC objects from CAS using $O(1)$ steps per operation [3]. See [14] for a survey of other implementations that use less space or handle larger LL/SC objects.

Many non-blocking implementations of primitives that access multiple objects use the cooperative technique, first described by Turek, Shasha and Prakash [19] and Barnes [5]. Instead of using locks that give a process exclusive access to a part of the data structure, this approach gives exclusive access to operations. If the process performing an operation that holds a lock is slow, other processes can help complete the operation and release the lock.

The cooperative technique was also used recently for a wait-free universal construction [8] and to obtain non-blocking binary search trees [10] and Patricia tries [16]. The approach used here is similar.

Israeli and Rappoport [13] used a version of the cooperative technique to implement multi-word CAS from single-word CAS (and sketched how this could be used to implement multi-word SC operations). However, their approach applies single-word CAS to very large words. The most efficient implementation of $k$-word CAS [18] first uses single-word CAS to replace each of the $k$ words with a pointer to a record containing information about the operation, and then uses single-word CAS to replace each of these pointers with the desired new value and update the status field of the record. In the absence of contention, this takes $2k + 1$ CAS steps. In contrast, in our implementation, an SCX that depends on LLXs of $k$ Data-records performs $k + 1$ single-word CAS steps when there is no contention, no matter how many words each record contains. So, our weaker primitives can be significantly more efficient than multi-word CAS or multi-word RMW [1, 4], which is even more general.

If $k$ Data-records are removed from a data structure by a multi-word CAS, then the multi-word CAS must depend on every mutable field of these records to avoid another process from concurrently updating any of them. It is possible to use $k$-word CAS to apply to $k$ Data-records instead of $k$ words with indirection: Every Data-record is represented by a single word containing a pointer to the contents of the record. To change any fields of the Data-record, a process needs to replace the pointer to a new copy of its contents containing the updated values. However, the extra level of indirection affects all reads, slowing them down considerably.

Luchangco, Moir and Shavit [15] defined the $k$-compare-single-swap (KCSS) primitive, which atomically tests whether $k$ specified memory locations contain specified values and, if all tests succeed, writes a value to one of the locations. They provided an obstruction-free implementation of KCSS, meaning that a process performing a KCSS is guaranteed to terminate if it runs alone. They implemented KCSS using an obstruction-free implementation of LL/SC from CAS. Specifically, to try to update location $v$ using KCSS, a process performs $LL(v)$, followed by two collects of the other $k − 1$ memory locations. If $v$ has its specified value, both collects return their specified values, and the contents of these memory locations do not change between the two collects, the process performs SC to change the value of $v$. Unbounded version numbers are used both in their implementation of LL/SC and to avoid the ABA problem between the two collects.

Our LLX and SCX primitives can be viewed as multi-Data-record-LL and single-Data-record-SC primitives, with the ability to finalize Data-records. We shall see that this extra ability is extremely useful for implementing pointer-based data structures. In addition, our implementation of LLX and SCX allows us to develop shared data structures that satisfy the non-blocking progress condition, which is stronger than obstruction-freedom.

3. THE PRIMITIVES

Our primitives operate on a collection of Data-records of various user-defined types. Each type of Data-record has a fixed number of mutable fields (each fitting into a single word), and a fixed number of immutable fields (each of which can be large). Each field is given a value when the Data-record is created. Fields can contain pointers that refer to other Data-records. Data-records are accessed using LLX, SCX and VLX, and reads of individual mutable or immutable fields of a Data-record. Reads of mutable fields are permitted because a snapshot of a Data-record’s fields is sometimes excessive, and it is sometimes sufficient (and more efficient) to use reads instead of LLXs.

An implementation of LL and SC from CAS has to ensure that, between when a process performs LL and when it next performs SC on the same word, the value of the word has not changed. Because the value of the word could change and
then change back to a previous value, it is not sufficient to check that the word has the same value when the LL and the SC are performed. This is known as the ABA problem. It also arises for implementations of LLX and SCX from CAS. A general technique to overcome this problem is described in Section 4.1. However, if the data structure designer can guarantee that the ABA problem will not arise (because each SCX never attempts to store a value into a field that previously contained that value), our implementation can be used in a more efficient manner.

Before giving the precise specifications of the behaviour of LLX and SCX, we describe how to use them, with the implementation of a multiset as a running example. The multiset abstract data type supports three operations: \text{Get}(\text{key}), which returns the number of occurrences of \text{key} in the set, \text{Insert}(\text{key}, \text{count}), which inserts \text{count} occurrences of \text{key} into the set, and \text{Delete}(\text{key}, \text{count}), which deletes \text{count} occurrences of \text{key} from the set and returns \text{True}, provided there are at least \text{count} occurrences of \text{key} in the set. Otherwise, it simply returns \text{False}.

Suppose we would like to implement a multiset using a sorted, singly-linked list. We represent each node in the list by a Data-record with an immutable field \text{key}, which contains a key in the multiset, and mutable fields: \text{count}, which records the number of times \text{key} appears in the multiset, and \text{next}, which points to the next node in the list. The first and last elements of the list are sentinel nodes with special keys $-\infty$ and $\infty$, respectively, which never occur in the multiset, and count 0.

Figure 5 shows how updates to the list are handled. Insertion behaves differently depending on whether the key is already present. Likewise, deletion behaves differently depending on whether it removes all copies of the key. For example, consider the operation \text{Delete}(d, 2) depicted in Figure 5(c). This operation removes node \text{r} by changing \text{p.next} to point to a new copy of \text{r.next}. A new copy is used to avoid the ABA problem, since \text{p.next} may have pointed to \text{r.next} in the past. To perform \text{Delete}(d, 2), a process first invokes LLXs on \text{p}, \text{r}, and \text{r.next}. Second, it creates a copy \text{r.next'} of \text{r.next}. Finally, it performs an SCX that depends on these three LLXs. This SCX attempts to change \text{p.next} to point to \text{r.next'}. This SCX will succeed only if none of \text{p}, \text{r} or \text{r.next} have changed since the aforementioned LLXs. Once \text{r} and \text{r.next} are removed from the list, we want subsequent invocations of LLX and SCX to be able to detect this, so that we can avoid, for example, erroneously inserting a key into a deleted part of the list. Thus, we specify in our invocation of SCX that \text{r} and \text{r.next} should be \text{finalized} if the SCX succeeds. Once a Data-record is finalized, it can never be changed again.

LLX (a pointer to) a Data-record \text{r} as its argument. Ordinarily, it returns either a snapshot of \text{r}'s mutable fields or \text{Finalized}. If an LLX(\text{r}) is concurrent with an SCX involving \text{r}, it is allowed to fail and return \text{Fail}. SCX takes four arguments: a sequence \text{V} of (pointers to) Data-records upon which the SCX depends, a subsequence \text{R} of \text{V} containing (pointers to) the Data-records to be finalized, a mutable field \text{fld} of a Data-record in \text{V} to be modified, and a value \text{new} to store in this field. VLX takes a sequence \text{V} of (pointers to) Data-records as its only argument. Each SCX and VLX and returns a Boolean value.

For example, in Figure 5(c), the \text{Delete}(d, 2) operation invokes SCX(\text{V}, \text{R}, \text{fld}, \text{new}), where \text{V} = (\text{p}, \text{r}, \text{r.next}), \text{R} = (\text{r}, \text{r.next'}), \text{fld} is the next pointer of \text{p}, and \text{new} points to the node \text{r.next'}.

A terminating LLX is called \text{successful} if it returns a snapshot or \text{Finalized}, and \text{unsuccessful} if it returns \text{Fail}. A terminating SCX or VLX is called \text{successful} if it returns \text{True}, and \text{unsuccessful} if it returns \text{False}. Our operations are wait-free, but an operation may not terminate if the process performing it fails, in which case the operation is neither successful nor unsuccessful. We say an invocation \text{I} of LLX(\text{r}) by a process \text{p} is \text{linked to} an invocation \text{I'} of SCX(\text{V}, \text{R}, \text{fld}, \text{new}) or VLX(\text{V}) by process \text{p} if \text{r} is in \text{V}, \text{I} returns a snapshot, and between \text{I} and \text{I'}, process \text{p} performs no invocation of LLX(\text{r}) or SCX(\text{V'}, \text{R'}, \text{fld'}, \text{new'}) and no unsuccessful invocation of VLX(\text{V'}), for any \text{V'} that contains \text{r}. Before invoking VLX(\text{V}) or SCX(\text{V}, \text{R}, \text{fld}, \text{new}), a process must \text{set up} the operation by performing an LLX(\text{r}) linked to the invocation for each \text{r} in \text{V}.

### 3.1 Correctness Properties

An implementation of LLX, SCX and VLX is \text{correct} if, for every execution, there is a linearization of all successful LLXs, all successful SCXs, a subset of the non-terminating SCXs, all successful VLXs, and all reads, such that the following conditions are satisfied.

**C1:** Each read of a field \text{f} of a Data-record \text{r} returns the last value stored in \text{f} by an SCX linearized before the read (or \text{f}'s initial value, if no such SCX has modified \text{f}).

**C2:** Each linearized LLX(\text{r}) that does not return \text{Finalized} returns the last value stored in each mutable field \text{f} of \text{r} by an SCX linearized before the LLX (or \text{f}'s initial value, if no such SCX has modified \text{f}).

**C3:** Each linearized LLX(\text{r}) returns \text{Finalized} if and only if it is linearized after an SCX(\text{V}, \text{R}, \text{fld}, \text{new}) with \text{r} in \text{R}.

**C4:** For each linearized invocation \text{I} of SCX(\text{V}, \text{R}, \text{fld}, \text{new}) or VLX(\text{V}), and for each \text{r} in \text{V}, no SCX(\text{V'}, \text{R'}, \text{fld'}, \text{new'}) with \text{r} in \text{V'} is linearized between the LLX(\text{r}) linked to \text{I} and \text{I}.

The first three properties assert that successful reads and LLXs return correct answers. The last property says that an invocation of SCX or VLX does not succeed when it should not. However, an SCX can fail if it is concurrent with another SCX that accesses some Data-record in common. LLX/SC also exhibits analogous failures in real systems. Our progress properties limit the situations in which this can occur.

### 3.2 Progress Properties

In our implementation, LLX, SCX and VLX are technically wait-free, but this is only because they may fail. So, we must state progress properties in terms of \text{successful} operations. The first progress property guarantees that LLXs on finalized Data-records succeed.

**P1:** Each terminating LLX(\text{r}) returns \text{Finalized} if it begins after the end of a successful SCX(\text{V}, \text{R}, \text{fld}, \text{new}) with \text{r} in \text{R} or after another LLX(\text{r}) has returned \text{Finalized}.

The next progress property guarantees non-blocking progress of invocations of our primitives.

**P2:** If operations are performed infinitely often, then operations succeed infinitely often. However, this progress property leaves open the possibility
that only LLXs succeed. So, we want an additional progress property:

P3: If SCX and VLX operations are performed infinitely often, then SCX or VLX operations succeed infinitely often.

Finally, the following progress property ensures that update operations that are built using SCX can be made non-blocking.

P4: If SCX operations are performed infinitely often, then SCX operations succeed infinitely often.

When the progress properties defined here are used to prove that an application built from the primitives is non-blocking, there is an important, but subtle point: an SCX can be invoked only after it has been properly set up by a sequence of LLXs. However, if processes repeatedly perform LLX on Data-records that have been finalized, they may never be able to invoke an SCX. One way to prevent this from happening is to have each process keep track of the Data-records it knows are finalized. However, in many natural applications, for example, the multiset implementation in Section 5, explicit bookkeeping can be avoided. In addition, to ensure that changes to a data structure can continue to occur, there must always be at least one non-finalized Data-record. For example, in our multiset, head is never finalized and, if a node is reachable from head by following next pointers, then it is not finalized.

Our implementation of LLX, SCX and VLX in Section 4 actually satisfies stronger progress properties than the ones described above. For example, a VLX(V) or SCX(V, R, fld, new) is guaranteed to succeed if there is no concurrent SCX(V′, R′, fld′, new′) such that V and V′ have one or more elements in common. However, for the purposes of the specification of the primitives, we decided to give progress guarantees that are sufficient to prove that algorithms that use the primitives are non-blocking, but weak enough that it may be possible to design other, even more efficient implementations of the primitives. For example, our specification would allow some spurious failures of the type that occur in common implementations of ordinary LL/SC operations (as long as there is some guarantee that not all operations can fail spuriously).

4. IMPLEMENTATION OF PRIMITIVES

The shared data structure used to implement LLX, SCX and VLX consists of a set of Data-records and a set of SCX-records. (See Figure 1.) Each Data-record contains user-defined mutable and immutable fields. It also contains a marked bit, which is used to finalize the Data-record, and an info field. The marked bit is initially False and only ever changes from False to True. The info field points to an SCX-record that describes the last SCX that accessed the Data-record. Initially, it points to a dummy SCX-record. When an SCX accesses a Data-record, it changes the info field of the Data-record to point to its SCX-record. While this SCX is active, the info field acts as a kind of lock on the Data-record, granting exclusive access to this SCX, rather than to a process. (To avoid confusion, we call this freezing, rather than locking, a Data-record.) We ensure that an SCX S does not change a Data-record for its own purposes while it is frozen for another SCX S'. Instead, S uses the information in the SCX-record of S' to help S' complete (successfully or unsuccessfully), so that the Data-record can be unfrozen. This cooperative approach is used to ensure progress.

An SCX-record contains enough information to allow any process to complete an SCX operation that is in progress. V, R, fld and new store the arguments of the SCX operation that created the SCX-record. Recall that R is a subsequence of V and fld points to a mutable field f of some Data-record r′ in V. The value that was read from f by the LLX(r′) linked to the SCX is stored in old. The SCX-record has one of three states, InProgress, Committed or Aborted, which is stored in its state field. This field is initially InProgress. The SCX-record of each SCX that terminates is eventually set to Committed or Aborted, depending on whether or not it successfully makes its desired update. The dummy SCX-record always has state = Aborted. The allFrozen bit, which is initially False, gets set to True after all Data-records in V have been frozen for the SCX. The values of state and allFrozen change in accordance with the diagram in Figure 7. The steps in the pseudocode that cause these changes are also indicated. The infoFields field stores, for each r in V, the value of r's info field that was read by the LLX(r) linked to the SCX.
can only be changed by a process helping the SCX that created it. Once a Data-record \( r \) is marked and \( r.info.state \) becomes Committed, \( r \) will never be modified again in any way. Figure 3 shows how a Data-record can change between frozen and unfrozen. The three bold boxes represent frozen Data-records. The other two boxes represent Data-records that are not frozen. A Data-record \( r \) can only become frozen when \( r.info \) is changed (to point to a new SCX-record whose state is InProgress). This is represented by the grey edges. The black edges represent changes to \( r.info.state \) or \( r.marked \). A frozen Data-record \( r \) can only become unfrozen when \( r.info.state \) is changed.

![Figure 3: Possible transitions for the marked field of a Data-record and the state of the SCX-record pointed to by the info field of the Data-record.](image)

### 4.1 Constraints

For the sake of efficiency, we have designed our implementation of LLX, VLX and SCX to work only if the primitives are used in a way that satisfies certain constraints, described in this section. We also describe general (but somewhat inefficient) ways to ensure these constraints are satisfied. However, there are often quite natural ways to ensure the constraints are satisfied without resorting to the extra work required by the general solutions.

Since our implementation of LLX, SCX and VLX uses helping to guarantee progress, each CAS of an SCX might be repeatedly performed by several helpers, possibly after the SCX itself has terminated. To avoid difficulties, we must show there is no ABA problem in the fields affected by these CAS steps.

The \( info \) field of a Data-record \( r \) is modified by CAS steps that attempt to freeze \( r \) for an SCX. All such steps performed by processes helping one invocation of SCX try to CAS the \( info \) field of \( r \) from the same old value to the same new value, and that new value is a pointer to a newly created SCX-record. Because the SCX-record is allocated a location that has never been used before, the ABA problem will not arise in the \( info \) field. (This approach is compatible with safe garbage collection schemes that only reuse an old address once no process can reach it by following pointers.)

A similar approach could be used to avoid the ABA problem in a mutable field of a Data-record: the new value could be placed inside a wrapper object that is allocated a new location in memory. (This is referred to as Solution 3 of the ABA problem in [9].) However, the extra level of indirection slows down accesses to fields.

To avoid the ABA problem, it suffices to prove the following constraint is satisfied.

- **Constraint:** For every invocation \( S \) of SCX(\( V, R \), \( fld, new \)), \( new \) is not the initial value of \( fld \) and no invocation of SCX(\( V', R', fld, new \)) was linearized before the LLX(\( r \)) linked to \( S \) was linearized, where \( r \) is the Data-record that contains \( fld \).

The multiset in Section 5 provides an example of a simple, more efficient way to ensure that this constraint is always satisfied.

To ensure property P4, we put a constraint on the way SCX is used. Our implementation of SCX(\( V, R, fld, new \)) does something akin to acquiring locks on each Data-record in \( V \). Livelock could occur if different invocations of SCX do not process Data-records in the same order. To prevent this, we could define a way of ordering all Data-records (for example, by their locations in memory) and each sequence passed to an invocation of SCX could be sorted using this ordering. However, this could be expensive. Moreover, to prove our progress properties, we do not require that all SCXs order their sequences \( V \) consistently. It suffices that, if all the Data-records stop changing, then the sequences passed to later invocations of SCX are all consistent with some total order. This property is often easy to satisfy in a natural way. More precisely, use of our implementation of SCX requires adherence to the following constraint.

- **Constraint:** Consider each execution that contains a configuration \( C \) after which the value of no field of any Data-record changes. There must be a total order on all Data-records created during this execution such that, if Data-record \( r_1 \) appears before Data-record \( r_2 \) in the sequence \( V \) passed to an invocation of SCX whose linked LLXs begin after \( C \), then \( r_1 \prec r_2 \).

For example, if one was using LLX and SCX to implement an unsorted singly-linked list, this constraint would be satisfied if the nodes in each sequence \( V \) occur in the order they are encountered by following next pointers from the beginning of the list, even if some operations could reorder the nodes in the list. While the list is changing, such a sequence may have repeated elements and might not be consistent with any total order.

### 4.2 Detailed algorithm description and sketch of proofs

Pseudocode for our implementation of LLX, VLX and SCX appears in Figure 4. If \( x \) contains a pointer to a record, then \( x.y := v \) assigns the value \( v \) to field \( y \) of this record, \( k;x.y \) denotes the address of this field and all other occurrences of \( x.y \) denote the value stored in this field.

**Theorem 1.** The algorithms in Figure 4 satisfy properties C1 to C4 and P1 to P4 in every execution where the constraints of Section 4.1 are satisfied.

The detailed proof of correctness [7] is quite involved, so we only sketch the main ideas here.

An LLX(\( r \)) returns a snapshot, Fail, or Finalized. At a high level, it works as follows. If the LLX determines that \( r \) is not frozen and \( r \)’s \( info \) field does not change while the LLX reads the mutable fields of \( r \), the LLX returns the values read as a snapshot. Otherwise, the LLX helps the SCX that has frozen \( r \) (if any), and returns Fail or Finalized. If the LLX returns Fail, it is not linearized. We now discuss in more detail how LLX operates and is linearized in the other two cases.

First, suppose the LLX(\( r \)) returns a snapshot at line 11. Then, the test at line 7 evaluates to True. So, either \( state = Aborted \), which means \( r \) is not frozen at line 5, or \( state = Committed \) and \( marked_r = False \). This also means \( r \) is not...
```
LLX(r) by process p
  ▷ Precondition: r ≠ Nil.
  marked1 := r.marked
  rinfo := r.info
  state := rinfo.state
  marked2 := r.marked
  if state = Aborted or (state = Committed and not marked2) then
    read r.m1, ..., r.my and record the values in local variables m1, ..., my
    if r.info = rinfo then
      store ⟨r, rinfo, ⟨m1, ..., my⟩⟩ in p’s local table
      return ⟨m1, ..., my⟩
    else
      if (rinfo.state = Committed or (rinfo.state = InProgress and HELP(rinfo))) and marked1 then
        return Finalized
      else
        if r.info.state = InProgress then HELP(r.info)
        return Fail

SCX(V, R, fld, new) by process p
  ▷ Preconditions: (1) for each r ∈ V, p has performed an invocation I_r of LLX(r) linked to this SCX
    (2) new is not the initial value of fld
    (3) for each r ∈ V, no SCX(V’, R’, fld, new) was linearized before I_r was linearized
    Let infoFields be a pointer to a newly created table in shared memory containing,
    for each r ∈ V, a copy of r’s info value in p’s local table of LLX results
    Let old be the value for fld stored in p’s local table of LLX results
    return HELP(pointer to new SCX-record(V, R, fld, new, old, InProgress, FALSE, infoFields))

HELP(scxPtr)
  ▷ Freeze all Data-records in scxPtr.V to protect their mutable fields from being changed by other SCXs
  for each r ∈ scxPtr.V enumerated in order do
    Let rinfo be the pointer indexed by r in scxPtr.infoFields
    if not CAS(r.info, rinfo, scxPtr) then
      ▷ freezing CAS
      if r.info ≠ scxPtr then
        ▷ Could not freeze r because it is frozen for another SCX
        if scxPtr.allFrozen = TRUE then
          ▷ the SCX has already completed successfully
          return TRUE
        else
          ▷ Atomically unfreeze all nodes frozen for this SCX
          scxPtr.state := Aborted
          return FALSE
      ▷ Finished freezing Data-records (Assert: state ∈ {InProgress, Committed})
      scxPtr.allFrozen := TRUE
      for each r ∈ scxPtr.R do r.marked := TRUE
      ▷ mark step
      CAS(scxPtr, fld, scxPtr.old, scxPtr.new)
      ▷ update CAS
      ▷ Finalize all r ∈ R, and unfreeze all r ∈ V \ R
      scxPtr.state := Committed
      ▷ commit step
    return TRUE

VLX(V) by process p
  ▷ Precondition: for each Data-record r ∈ V, p has performed an LLX(r) linked to this VLX
  for each r ∈ V do
    Let rinfo be the info field for r stored in p’s local table of LLX results
    if rinfo ≠ r.info then return FALSE  ▷ r changed since LLX(r) read info
    return TRUE  ▷ At some point during the loop, all r ∈ V were unchanged
```

Figure 4: Pseudocode for LLX, SCX and VLX.
frozen at line 5, since \( r.marked \) cannot change from \text{True} to \text{False}. The \text{LLX} reads \( r \)'s mutable fields (line 8) and rereads \( r.info \) at line 9, finding it the same as on line 4. In Section 4.1, we explained why this implies that \( r.info \) did not change between lines 4 and 9. Since \( r \) is not frozen at line 5, we know from Figure 3 that \( r \) is unfrozen at all times between line 5 and 9. We prove that mutable fields can change only while \( r \) is frozen, so the values read by line 8 constitute a snapshot of \( r \)'s mutable fields. Thus, we can linearize the \text{LLX} at line 9.

Now, suppose the \text{LLX}(\( r \)) returns \text{FINALIZED}. Then, the test on line 12 evaluated to \text{True}. In particular, \( r \) was already marked when line 3 was performed. If \( r.info.state = \text{InProgress} \) when line 12 was performed, \text{HELP}(\( r.info \)) was called and returned \text{True}. Below, we argue that \( r.info.state \) was changed to Committed before the return occurred. By Figure 3(a), the state of an \text{SCX}-record never changes after it is set to Committed. So, after line 12, \( r.info.state = \text{Committed} \) and, thus, \( r \) has been finalized. Hence, the \text{LLX} can be linearized at line 13.

When a process performs an \text{SCX}, it first creates a new \text{SCX}-record and then invokes \text{HELP} (line 21). The \text{HELP} routine performs the real work of the \text{SCX}. It is also used by a process to help other processes complete their \text{SCXs} (successfully or unsuccessfully). The values in an \text{SCX}-record's \text{old} and \text{infoFields} come from a table in the local memory of the process that invokes the \text{SCX}, which stores the results of the last \text{LLX} it performed on each \text{Data-record}. (In practice, the memory required for this table could be greatly reduced when a process knows which of these values are needed for future \text{SCXs}.)

Consider an invocation of \text{HELP}(\( U \)) by process \( p \) to carry out the work of the invocation \( S \) of \text{SCX}(\( V, R, fld, new \)) that is described by the \text{SCX}-record \( U \). First, \( p \) attempts to freeze each \( r \in V \) by performing a freezing CAS to store a pointer to \( U \) in \( r.info \) (line 26). Process \( p \) uses the value read from \( r.info \) by the \text{LLX}(\( r \)) linked to \( S \) as the old value for this \text{CAS} and, hence, it will succeed only if \( r \) has not been frozen for any other \text{SCX} since then. If \( p \)'s freezing \text{CAS} fails, it checks whether some other helper has successfully frozen the \text{Data-record} with a pointer to \( U \) (line 27).

If every \( r \in V \) is successfully frozen, \( p \) performs a frozen \text{step} to set \text{U.allFrozen} to \text{True} (line 37). After this frozen step, the \text{SCX} is guaranteed not to fail, meaning that no process will perform an abort step while helping this \text{SCX}. Then, for each \( r \in R \), \( p \) performs a mark \text{step} to set \( r.marked \) to \text{True} (line 38) and, from Figure 3, \( r \) remains frozen from then on. Next, \( p \) performs an update \text{CAS}, storing \text{new} in the field pointed to by \( fld \) (line 39), if successful. We prove that, among all the update \text{CAS} steps on \( fld \) performed by the helpers of \( U \), only the first can succeed. Finally, \( p \) unfreezes all \( r \in V \setminus R \) by performing a commit \text{step} that changes \( U.state \) to Committed (line 41).

Now suppose that, when \( p \) performs line 27, it finds that some \text{Data-record} \( r \in V \) is already frozen for another invocation \( S' \) of \text{SCX}. If \text{U.allFrozen} is \text{False} at line 29, then we can prove that no helper of \( S \) will ever reach line 37, so \( p \) can abort \( S \). To do so, it unfreezes each \( r \in V \) that it has frozen by performing an abort \text{step}, which changes \( U.state \) to Aborted (line 34), and then returns \text{False} (line 35) to indicate that \( S \) has been aborted. If \text{U.allFrozen} is \text{True} at line 29, it means that each element of \( V \), including \( r \), was successfully frozen by some helper of \( S \) and then, later, a process \( r \) for \( S' \). Since \( S \) cannot be aborted after \text{U.allFrozen} was set to \text{True}, its state must have changed from \text{InProgress} to \text{Committed} before \( r \) was frozen for another \text{SCX}-record. Therefore, \( S \) was successfully completed and \( p \) can return \text{True} at line 31.

We linearize an invocation of \text{SCX} at the first update \text{CAS} performed by one of its helpers. We prove that this update \text{CAS} always succeeds. Thus, all \text{SCXs} that return \text{True} are linearized, as well as possibly some non-terminating \text{SCXs}. The first update \text{CAS} of \text{SCX}(\( V, R, fld, new \)) modifies the value of \( fld \), so a read(\( fld \)) that occurs immediately after the update \text{CAS} will return the value of \text{new}. Hence, the linearization point of an \text{SCX} must occur at its first update \text{CAS}. There is one subtle issue about this linearization point: If an \text{LLX}(\( r \)) is linearized between the update \text{CAS} and commit step of an \text{SCX} that finalizes \( r \), it might not return \text{FINALIZED}, violating condition C3. However, this cannot happen, because, before the \text{LLX} is linearized on line 13, the \text{LLX} either sees that the commit step has been performed or helps the \text{SCX} perform its commit step.

An invocation \( I \) of \text{VLX}(\( V \)) is executed by a process \( p \) after \( p \) has performed an invocation of \text{LLX}(\( r \)) linked to \( I \), for each \( r \in V \). \text{VLX}(\( V \)) simply checks, for each \( r \in V \), that the info field of \( r \) is the same as when it was read by \( p \)'s last \text{LLX}(\( r \)) and, if so, \text{VLX}(\( V \)) returns \text{True}. In this case, we prove that each \text{Data-record} in \( V \) does not change between the linked \text{LLX} and the time its info field is reread. Thus, the \text{VLX} can be linearized at the first time it executes line 47. Otherwise, the \text{VLX} returns \text{False} to indicate that the \text{LLX} results may not constitute a snapshot.

We remark that our use of the cooperative method avoids costly recursive helping. If, while \( p \) is helping \( S \), it cannot freeze all of \( S \)'s \text{Data-records} because one of them is already frozen for a third \text{SCX}, \( p \) will simply perform an abort step, which unfreezes all \text{Data-records} that \( S \) has frozen. We briefly sketch why the progress properties described in Section 3.2 are satisfied. It follows easily from the code that an invocation of \text{LLX}(\( r \)) returns \text{FINALIZED} if it begins after the end of an \text{SCX} that finalized \( r \) or another \text{LLX} sees that \( r \) is finalized. To prove the progress properties P2, P3 and P4, we consider two cases.

First, consider an execution where only a finite number of \text{SCXs} are invoked. Then, only finitely many \text{SCX-records} are created. Each process calls \text{HELP}(\( U \)) if it sees that \( U.state = \text{InProgress} \), which it can do at most once for each \text{SCX-record} \( U \). Since every \text{CAS} is performed inside the \text{HELP} routine, there is some point after which no process performs a \text{CAS}, calls \text{HELP}, or sees a \text{SCX-record} whose \text{state} is \text{InProgress}. A \text{VLX} can fail only when an info field is modified by a concurrent operation and an \text{LLX} can only fail for the same reason or when it sees a \text{SCX-record} whose \text{state} is \text{InProgress}. Therefore, all \text{LLXs} and \text{VLXs} that begin after this point will succeed, establishing P2 and P3. Moreover, P4 is vacuously satisfied in this case.

Now, consider an execution where infinitely many \text{SCXs} are invoked. To derive a contradiction, suppose only finitely many \text{SCXs} succeed. Then, there is a time after which no more \text{SCXs} succeed. The constraint on the sequences passed to invocations of \text{SCXs} ensures that all \text{SCXs} whose linked \text{LLXs} begin after this time will attempt to freeze their sequences of \text{Data-records} in a consistent order. Thus, one of these \text{SCXs} will succeed in freezing all of the \text{Data-records} that were passed to it and will successfully complete. This
is a contradiction. Thus, infinitely many of the SCXs do succeed, establishing properties P2, P3 and P4.

4.3 Additional Properties

Our implementation of SCX satisfies some additional properties, which are helpful for designing certain kinds of nonblocking data structures so that query operations can run efficiently. Consider a pointer-based data structure with a fixed set of Data-records called entry points. An operation on the data structure starts at an entry point and follows pointers to visit other Data-records. (For example, in our multiset example, the head of the linked list is the sole entry point for the data structure.) Thus, we say that a Data-record is in the data structure if it can be reached by following pointers from an entry point, and a Data-record r is removed from the data structure by an SCX if r is in the data structure immediately prior to the linearization point of the SCX and is not in the data structure immediately afterwards.

If the data structure is designed so that a Data-record is finalized when (and only when) it is removed from the data structure, then we have the following additional properties.

PROPOSITION 2. Suppose each linearized SCX(V, R, fld, new) removes precisely the Data-records in R from the data structure.

- If LLX(r) returns a value different from Fail or Finalized, r is in the data structure just before the LLX is linearized.
- If an SCX(V, R, fld, new) is linearized and new is (a pointer to) a Data-record, then this Data-record is in the data structure just after the SCX is linearized.
- If an operation reaches a Data-record r by following pointers read from other Data-records, starting from an entry point, then r was in the data structure at some earlier time during the operation.

The first two properties are straightforward to prove. The last property is proved by induction on the Data-records reached. For the base case, entry points are always reachable. For the induction step, consider the time when an operation reads a pointer to r from another Data-record r’ that the operation reached earlier. By the induction hypothesis, there was an earlier time t during the operation when r’ was in the data structure. If r’ already contained a pointer to r at t, then r was also in the data structure at that time. Otherwise, an SCX wrote a pointer to r in r’ after t, and just after that update occurred, r’ and r were in the data structure (by the second part of the proposition).

The last property is a particularly useful one for linearizing query operations. It means that operations that search through a data structure can use simple reads of pointers instead of the more expensive LLX operations. Even though the Data-record that such a search operation reaches may have been removed from the data structure by the time it is reached, the lemma guarantees that there was a time during the search when the Data-record was in the data structure. For example, we use this property to linearize searches in our multiset algorithm in Section 5.

5. AN EXAMPLE: MULTISET

We now give a detailed description of the implementation of a multiset using LLX and SCX that was introduced in Section 3. We assume that keys stored in the multiset are drawn from a totally ordered set. Each key of the multiset is stored in a Data-record called a Node along with a count (see Figure 6). The Nodes are arranged in a singly linked list sorted by keys. To avoid special cases, we have two sentinel nodes, head and tail, at the beginning and end of the list. These sentinel nodes have keys $-\infty$ and $\infty$, where $-\infty < k < \infty$ for every other possible key k. Pseudocode is presented in Figure 6. Search(key) traverses the list, starting from the head, by reading next pointers until reaching the first node r whose key is at least key. This node and the preceding node p are returned. Get(key) performs Search(key), outputs r’s count if r’s key matches key, and outputs 0, otherwise.

An invocation I of Insert(key, count) starts by calling Search(key). Using the nodes p and r that are returned, it updates the data structure. It decides whether key is already in the multiset (by checking whether r.key = key) and, if so, it invokes LLX(r) followed by an SCX linked to r to increase r.count by count, as depicted in Figure 5(b). Otherwise, I performs the update depicted in Figure 5(a): It invokes LLX(p), checks that p still points to r, creates a node, new, and invokes an SCX linked to p to insert new between p and r. If p no longer points to r, the LLX returns Fail or Finalized, or the SCX returns False, then I restarts.

An invocation I of Delete(key, count) also begins by calling Search(key). It invokes LLX on the nodes p and r checks that p still points to r. If r does not contain at least count copies of key, then I returns False. If r contains ex-
Start

Each Get and each Delete that returns False is linearized at the linearization point of the Search it performs. Every other Insert or Delete is linearized at its successful SCX. Linearity of all operations then follows from the next invariant.

Lemma 5. At every time \( t \), the multiset of keys in the data structure is equal to the multiset of keys that would result from the atomic execution of the sequence of operations linearized up to time \( t \).

To prove the algorithm is non-blocking, suppose there is some infinite execution in which only finitely many operations terminate. Then, eventually, no more Insert or Delete operations perform a successful SCX, so there is a time after which the pointers that form the linked list stop changing. This implies that all calls to the Search subroutine must terminate. Since a Get operation merely calls Search, all Get operations must also terminate. Thus, there is some collection of Insert and Delete operations that take steps forever without terminating. We show that such operation sets up and performs an SCX infinitely often. For any Insert or Delete operation, consider an any iteration of the loop that begins after the last successful SCX changes the list. By Lemma 4 and Proposition 2, the nodes \( p \) and \( r \) reached by the Search in that iteration were in the data structure at some time during the Search and, hence, throughout the Search. So when the Insert or Delete performs LLXs on \( p \) or \( r \), they cannot return Finalized. Moreover, they must succeed infinitely often by property P2, and this allows the Insert or Delete to perform an SCX infinitely often. By property P4, SCXs will succeed infinitely often, a contradiction.
Thus, we have the following theorem.

**Theorem 6.** The algorithms in Figure 6 implement a non-blocking, linearizable multiset.

### 6. CONCLUSION

The LLX, SCX and VLX primitives we introduce in this paper can also be used to produce practical, non-blocking implementations of a wide variety of tree-based data structures. In [6], we describe a general method for obtaining such implementations and use it to design a provably correct, non-blocking implementation of a chromatic tree, which is a relaxed variant of a red-black tree. Furthermore, we provide an experimental performance analysis, comparing our Java implementation of the chromatic search tree to leading concurrent implementations of dictionaries. This demonstrates that our primitives enable efficient non-blocking implementations of more complicated data structures to be built (and added to standard libraries), together with manageable proofs of their correctness.

Our implementation of LLX, SCX and VLX relies on the existence of efficient garbage collection, which is provided in managed languages such as Java and C++. However, in other languages, such as C++, memory management is an issue. This can be addressed by a new, efficient memory reclamation technique for non-blocking trees. Manuscript available from http://www.cs.utoronto.ca/~tabrown.

Acknowledgements.

Funding was provided by the Natural Sciences and Engineering Research Council of Canada.

### 7. REFERENCES


APPENDIX

A. COMPLETE PROOF

A.1 Basic properties

We begin with some elementary properties that are needed to prove basic lemmas about freezing. In particular, we show that the info field of a Data-record cannot experience an ABA problem.

**Definition 7.** Let $I'$ be an invocation of SCX($V$, $R$, $fld$, new) or VLX($V$) by a process $p$, and $r$ be a Data-record in $V$. We say an invocation $I$ of LLX($r$) is linked to $I'$ if and only if:
1. $I$ returns a value different from Fail or Finalized, and
2. no invocation of LLX($r$), SCX($V'$, $R'$, $fld'$, new'), or VLX($V'$), where $V'$ contains $r$, is performed by $p$ between $I$ and $I'$.

**Observation 8.** An SCX-record $U$ created by an invocation $S$ of SCX satisfies the following invariants.
1. $U.fld$ points to a mutable field $f$ of a Data-record $r'$ in $U.V$. The value stored in $U.old$ was read at line 8 from $f$ by the LLX($r'$) linked to $S$.
2. For each $r$ in $U.V$, the pointer indexed by $r$ in $U.infoFields$ was read from $r.info$ at line 4 by the LLX($r$) linked to $S$.
3. For each $r$ in $U.V$, the LLX($r$) linked to $S$ must enter the if-block at line 7, and see $r.info = r.info = r.info$ at line 9.

**Proof.** None of the fields of an SCX-record except state change after they are initialized at line 21. The contents of the table pointed to by $U.infoFields$ do not change, either. Therefore, it suffices to show that these invariants hold when $u$ is created. The proof of these invariants follows immediately from the precondition of SCX, the pseudocode of LLX, and the definition of an LLX linked to an SCX.

The following two definitions ease discussion of the important steps that access shared memory.

**Definition 9.** A process is said to be helping an invocation of SCX that created an SCX-record $U$ whenever it is executing HELP($ptr$), where $ptr$ points to $U$. (For brevity, we sometimes say a process is “helping $U” instead of “helping the SCX that created $U”.”)

Note that, since HELP does not call itself directly or indirectly, a process cannot be helping two different invocations of SCX at the same time.

**Definition 10.** We say that a freezing CAS, update CAS, frozen step, mark step, abort step, commit step or frozen check step $S$ belongs to an SCX-record $U$ when $S$ is performed by a process helping $U$. We say that a frozen step, mark step, abort step, commit step or frozen check step is successful if it changes the field it modifies to a different value. A freezing CAS or update CAS is successful if the CAS succeeds. Any step is unsuccessful if it is not successful.

**Lemma 11.** No freezing CAS, update CAS, frozen step, mark step, abort step, commit step or frozen check step belongs to the dummy SCX-record.

**Proof.** According to Definition 10, we must simply show that no process ever helps the dummy SCX-record $D$. To derive a contradiction, assume there is some invocation of HELP($ptr$) where $ptr$ points to $D$, and let $H$ be the first such invocation. HELP is only invoked at lines 12, 15 and 21.

If $H$ occurs at line 12 then $D.state = InProgress$ at some point before $H$ (by line 12). Since $D.state$ is initially Aborted, and InProgress is never written into any state field, this is impossible.

Now, suppose $H$ occurs at line 15. If $r.info$ points to $D$ both times it is read at line 15, then we obtain the same contradiction as the previous case. Otherwise, a successful freezing CAS changes $r.info$ to point to $D$ in between the two reads of $r.info$. By line 26, this freezing CAS must occur in an invocation of HELP($ptr$). However, since this freezing CAS must precede the first invocation, $H$, of HELP($ptr$), this case is impossible.

$H$ cannot occur at line 21, since that line calls HELP on a newly created SCX-record, not $D$. $lacklozenge$

**Lemma 12.** Every update to the info field of a Data-record $r$ changes $r.info$ to a value that has never previously appeared there. Hence, there is no ABA problem on info fields.

**Proof.** We first note that $r.info$ can only be changed by a freezing CAS at line 26. When a freezing CAS attempts to change an info field $f$ from $x$ to $y$, $y.infoFields$ contains $x$ (by line 25). Then, since $y.infoFields$ does not change after the SCX-record pointed to by $y$ is created, the SCX-record pointed to by $x$ was created before the SCX-record pointed to by $y$. So, letting $a_1, a_2, ...$ be the sequence of SCX-records ever pointed to by $r.info$, we know that $a_1, a_2, ...$, were created (at line 21) in that order. Since we have assumed memory allocations always receive new addresses, $a_1, a_2, ...$ are distinct. $lacklozenge$

**Definition 13.** A freezing CAS on a Data-record $r$ is one that operates on $r.info$. A mark step on a Data-record $r$ is one that writes to $r.marked$.

**Lemma 14.** For each Data-record $r$ in the $V$ sequence of an SCX-record $U$, only the first freezing CAS belonging to $U$ on $r$ can succeed.

**Proof.** Let $ptr$ be a pointer to $U$, and $fcas$ be the first freezing CAS belonging to $U$ on $r$. Let $r.info$ be the old value used by $fcas$. By Definition 10, the new value used by $fcas$ is $ptr$. Since $fcas$ belongs to $U$, Lemma 11 implies that $U$ is not the dummy SCX-record initially pointed to by each info field. Hence, $U$ was created by an invocation of SCX, so Observation 8.3 implies that $r.info contained r.info during the LLX($r$) linked to $S$. Since the LLX($r$) linked to $S$ terminates before the start of $S$, and $S$ creates $U$, the LLX($r$) linked to $S$ must terminate before any invocation of HELP($ptr$) begins. From the code of HELP, $fcas$ occurs in an invocation of HELP($ptr$). Thus, $r.info$ contains $r.info$ at some point before $fcas$. If $fcas$ is successful, then $r.info$ contains $r.info$ just before $fcas$, and $ptr$ just after. Otherwise, $r.info$ contains $r.info$ at some point before $fcas$, but contains some other value just before $fcas$. In either case, Lemma 12 implies that $r.info$ can never again contain $r.info$ after $fcas$. Finally, since each freezing CAS belonging to $U$ on $r$ uses $r.info$ as its old value (by line 25 and the fact that table $U.infoFields$ does not change after it is first created), there can be no successful freezing CAS belonging to $U$ on $r$ after $fcas$. $lacklozenge$
A.2 Changes to the info field of a Data-record and the state field of an SCX-record

We prove that freezing of nodes proceeds an orderly way. The first lemma shows that a process cannot freeze a node that is frozen by a different operation that is still in progress.

**Lemma 15.** The info field of a Data-record $r$ cannot be changed while $r$.info points to an SCX-record with state InProgress.

**Proof.** Suppose an info field of a Data-record $r$ is changed while it points to an SCX-record $U$ with $U.state = InProgress$. This change can only be performed by a successful freezing CAS $fcas$ whose old value is a pointer to $U$ and whose new value is a pointer to $W$. Let $S$ be the invocation of SCX that created $W$. From line 25, we can see that the old value for $feas$ (a pointer to $U$) is stored in the table $W.infoFields$ and, by Observation 8.3, this value was read from $r$.info (at line 4) by the LLX($r$) linked to $S$. Hence, the LLX($r$) linked to $S$ reads $U.state$ at line 5. By Observation 8.4, the LLX($r$) linked to $S$ passes the test at line 7 and enters the if-block. This implies that, when $U.state$ was read at line 5, either it was Committed and $r$ was unmarked, or it was Aborted. Thus, $U.state$ must be Aborted or Committed prior to $fcas$, and the claim follows from the fact that InProgress is never written to $U.state$. $\square$

It follows easily from Lemma 15 that if a node is frozen for an operation, it remains so until the operation is committed or aborted.

**Lemma 16.** If there is a successful freezing CAS $fcas$ belonging to an SCX-record $U$ on a Data-record $r$, and some time $t$ after the first freezing CAS belonging to $U$ on $r$ and before the first abort step or commit step belonging to $U$, then $r$.info points to $U$ at $t$.

**Proof.** Since $fcas$ belongs to $U$, by Lemma 11, $U$ cannot be the dummy SCX-record, so $U$ is created at line 21, where $U.state$ is initially set to InProgress. Let $t'$ be the time the first abort step or commit step belonging to $U$ on $r$ occurs. By Lemma 14, $fcas$ must be the first freezing CAS belonging to $U$ on $r$. Thus, $t$ is after $fcas$ occurs, and before $t'$. Immediately following $fcas$, $r$.info points to $U$. From the code, $U.state$ can only be changed by an abort step or commit step belonging to $U$. Therefore, $U.state = InProgress$ at all times after $fcas$ and before $t'$. By Lemma 15, $r$.info cannot change after $fcas$, and before $t'$. Hence, $r$.info points to $U$ at $t$. $\square$

A frozen step occurs only after all Data-records are successfully frozen.

**Lemma 17.** If a frozen step belongs to an SCX-record $U$ then, for each $r$ in $U.V$, there is a successful freezing CAS belonging to $U$ on $r$ that occurs before the first frozen step belonging to $U$.

**Proof.** Suppose a frozen step belongs to $U$. Let $fstep$ be the first such frozen step and let $H$ be the invocation of HELP that performs $fstep$. Since $fstep$ occurs at line 37, for each Data-record $r$ in $U.V$, $H$ must perform a successful freezing CAS belonging to $U$ on $r$ or see $other Ptr = scxPtr$ in the preceding loop. If $H$ performs a successful freezing CAS belonging to $U$ on $r$, then we are done. Otherwise, $r$.info = scxPtr at some point before $fstep$. Since $fstep$ belongs to $U$, scxPtr points to $U$. From Lemma 11, $U$ is not the dummy SCX-record to which $r$.info initially points. Hence, some process must have changed $r$.info to point to $U$, which can only be done by a successful freezing CAS. $\square$

Finally, we show that Data-records are frozen in the correct order. (This will be useful later on to show that no livelock can occur.)

**Lemma 18.** Let $U$ be an SCX-record, and $(r_1, r_2, ..., r_l)$ be the sequence of Data-records in $U.V$. For $i \geq 2$, a freezing CAS belonging to $U$ on Data-record $r_i$ can occur only after a successful freezing CAS belonging to $U$ on $r_{i-1}$.

**Proof.** Let $fcas$ be a freezing CAS belonging to $U$ on $r_i$, for some $i \geq 2$. Let $H$ be the invocation of HELP which performs $fcas$. The loop in $H$ iterates over the sequence $r_1, r_2, ..., r_i$, so $H$ performs $fcas$ in iteration $i$ of the loop. Since $H$ reaches iteration $i$, $H$ must perform iteration $i - 1$. Thus, by the code of HELP, $H$ must perform a freezing CAS $feas'$ belonging to $U$ on $r_{i-1}$ at line 26 before $fcas$. If $feas'$ succeeds, then the claim is proved. Otherwise, $H$ will check whether $r_{i-1}.info$ is equal to scxPtr at line 27. Since HELP does not return in iteration $i - 1$, $r_{i-1}.info = scxPtr$. This can only be true if $U$ is the dummy SCX-record, or there has already been a successful freezing CAS belonging to $U$ on $r_{i-1}$. Since $feas$ belongs to $U$, Lemma 11 implies that $U$ cannot be the dummy SCX-record. $\square$

A.3 Proving state and info fields change as described in Fig. 7 and Fig. 3(a)

In this section, we first prove that an SCX-record’s state transitions respect Figure 7, then we expand upon the information in Figure 7 by showing that frozen steps, abort steps, commit steps and successful freezing CASs proceed as illustrated in Figure 3(a).

We now prove an SCX-record $U$‘s state transitions respect Figure 7 by noting that a $U.state$ is never changed to InProgress, and proving that it does not change from Aborted to Committed or vice versa. Since $U.state$ can only be changed by a commit step or abort step belonging to $U$, and each commit step is preceded by a frozen step (by the code of HELP), it suffices to show that there cannot be both a frozen step and an abort step belonging to an SCX-record.

**Lemma 19.** Let $U$ be an SCX-record. Suppose that there is a successful freezing CAS belonging to $U$ on each Data-record in $U.V$. Then, a frozen check step belonging to $U$ cannot occur until after a frozen step belonging to $U$ has occurred.

**Proof.** To derive a contradiction, suppose that the first frozen check step $fstep$ belonging to $U$ occurs before any frozen step belonging to $U$. Let $H$ be the invocation of HELP in which $fstep$ occurs. Before $fstep$, $H$ performs
The next five lemmas prove that any successful freezing CASs belonging to an SCX-record \(U\) must occur prior to the first frozen step or abort step belonging to \(U\). This result allows us to fill in the gaps between Figure 7 and Figure 3(a).

**Lemma 24.** Let \(U\) be an SCX-record, and let \(\langle r_1, r_2, \ldots, r_l \rangle\) be the sequence of Data-records in \(U\). Suppose an abort step belongs to \(U\) and let \(astep\) be the first such abort step. Then, there is a \(k \in \{1, \ldots, l\}\) such that

1. a freezing CAS belonging to \(U\) on \(r_k\) occurs prior to \(astep\),
2. there is no successful freezing CAS belonging to \(U\) on \(r_k\),
3. for each \(i \in \{1, \ldots, k - 1\}\), a successful freezing CAS belonging to \(U\) on \(r_i\) occurs prior to \(astep\), and no successful freezing CAS belonging to \(U\) on \(r_i\) occurs after \(astep\), and
4. \(r_k\) info changes after the \(\text{LLX}(r_k)\) linked to \(S\) reads \(r_k\) info at line 9 and before the first freezing CAS belonging to \(U\) on \(r_k\).

**Proof.** Let \(H\) be the invocation of HELP that performs \(astep\) and \(k\) be the iteration of the loop in HELP during which \(H\) performs \(astep\). The loop in \(H\) iterates over the sequence \(r_1, r_2, \ldots, r_l\) of Data-records.

Claim 1 follows from the definition of \(k\). Before \(H\) performs \(astep\), it performs a freezing CAS belonging to \(H\) on \(r_k\) at line 26.

To derive a contradiction, suppose Claim 2 is false, i.e., there is a successful freezing CAS belonging to \(U\) on \(r_k\). By Claim 1, \(feas\) is before \(astep\). From Corollary 22 and the fact that \(astep\) occurs, we know that no commit step belongs to \(U\). By Lemma 16, \(r_k\) info points to \(U\) at all times between the first freezing CAS belonging to \(U\) on \(r_k\) and \(astep\). However, this contradicts the fact that \(r_k\) info does not point to \(U\) when \(H\) performs line 27 just before performing \(astep\).

We now prove Claim 3. By Claim 1, prior to \(astep\), \(H\) performs a freezing CAS belonging to \(U\) on \(r_k\). By Lemma 18, this can only occur after a successful freezing CAS belonging to \(U\) on \(r_i\) for all \(i < k\). By Lemma 14, there is no successful freezing CAS belonging to \(U\) on \(r_i\) after \(astep\).

We now prove Claim 4. By Claim 1 and Claim 2, an unsuccessful freezing CAS \(feas\) belonging to \(U\) on \(r_k\) occurs prior to \(astep\). By line 25 and Observation 8.3, the old value for \(feas\) is read from \(r_k\) info and stored in \(rinfo\) at line 4 by the \(\text{LLX}(r_k)\) linked to \(S\). By Observation 8.4, the \(\text{LLX}(r_k)\) linked to \(S\) again sees \(r_k\) info = \(rinfo\) at line 9. Thus, since \(feas\) fails, \(r_k\) info must change after the \(\text{LLX}(r_k)\) linked to \(S\) performs line 9 and before \(feas\) occurs.

**Lemma 25.** No freezing CAS belonging to an SCX-record \(U\) is successful after the first frozen step or abort step belonging to \(U\).

**Proof.** First, suppose a frozen step belongs to \(U\), and let \(fstep\) be the first such frozen step. Then, by Lemma 17, there is a successful freezing CAS belonging to \(U\) on \(r\) for each \(r\) in \(U\).V that occurs before \(fstep\). By Lemma 14, only the first freezing CAS belonging to \(U\) on \(r\) can be successful. Hence, no freezing CAS belonging to \(U\) is successful after \(fstep\).

Now, suppose an abort step belongs to \(U\), and let \(astep\) be the first such abort step. Let \(\langle r_1, r_2, \ldots, r_l \rangle\) be the sequence of Data-records in \(U\). By Lemma 24, there is a
of $k \in \{1,\ldots,l\}$ such that no successful freezing CAS belonging to $U$ is performed on any $r_i \in \{r_1,\ldots,r_{k-1}\}$ after $a_{step}$, and no successful freezing CAS belonging to $U$ on $r_k$ ever occurs. By Lemma 18, there is no freezing CAS belonging to $U$ on any $r_i \in \{r_{k+1},\ldots,r_l\}$. □

**Corollary 26.** If there is a successful freezing CAS $f_{cas}$ belonging to an SCX-record $U$ on a Data-record $r$, then $f_{cas}$ occurs before time $t$, when the first abort step or commit step belonging to $U$ occurs. Moreover, $r.info$ points to $U$ at all times after $f_{cas}$ occurs, and before $t$.

**Proof.** By Lemma 25, $f_{cas}$ occurs before $t$. The claim then follows from Lemma 16. □

**Lemma 27.** Changes to the state and allFrozen fields of an SCX-record, as well as frozen steps, abort steps, commit steps and successful freezing CASs can only occur as depicted in Figure 3(a).

**Proof.** Initially, the dummy SCX-record has $state = \text{Aborted}$ and allFrozen = False and, by Lemma 11, they never change.

Every other SCX-record $U$ initially has $state = \text{InProgress}$ and allFrozen = False. Only abort steps, frozen steps, and commit steps can change $state$ or allFrozen. From the code of Help, each transition shown in Figure 3(a), results in the indicated values for $state$ and allFrozen. A commit step on line 41 must be preceded by a frozen step on line 37. Therefore, from [InProgress, False], the only outgoing transitions are to [Aborted, False] and [InProgress, True]. By Lemma 21, there cannot be both a frozen step and an abort step belonging to $U$. Hence, from [Aborted, False], there cannot be a frozen step or commit step and there cannot be an abort step from [InProgress, True] or [Committed, True].

By Lemma 25, successful freezing CASs can only occur when $state = \text{InProgress}$ and allFrozen = False. From the code of Help, for each $r$ in $U.R$, the first mark step belonging to $U$ on $r$ must occur after the first frozen step belonging to $U$ and before the first commit step belonging to $U$. Since each $r$ in $U.R$ initially has $r.marked = \text{False}$, and $r.marked$ is only changed at line 38, where it is set to True, only the first mark step belonging to $U$ on $r$ can be successful. □

### A.4 The period of time over which a Data-record is frozen

We now prove several lemmas which characterize the period of time over which a Data-record is frozen for an SCX-record. We first use the fact that the state of an SCX-record cannot change from Aborted to Committed to extend Lemma 15 to prove that the info field of a Data-record cannot be changed while the Data-record is frozen for an SCX-record. In the following, the phrase “after $X$ and before the first time $Y$ happens” should be interpreted to mean “after $X$” in the event that $Y$ never happens.

**Lemma 28.** If a frozen step belongs to an SCX-record $U$ then, for each $r$ in $U.V$, a freezing CAS belonging to $U$ on $r$ precedes the first frozen step belonging to $U$, and $r$ is frozen for $U$ at all times after the first freezing CAS belonging to $U$ on $r$ and before the first commit step belonging to $U$.

**Proof.** Fix any $r$ in $U.V$. If a frozen step belongs to $U$ then, by Lemma 17, it is preceded by a successful freezing CAS belonging to $U$ on $r$. Further, by Lemma 21, no abort step belongs to $U$. Thus, by Corollary 26, $r.info$ points to $U$ at all times between time $t_0$, when the first freezing CAS belonging to $U$ on $r$ occurs, and time $t_1$, when the first commit step belonging to $U$ occurs (after the first frozen step). Since no abort step belongs to $U$. $U.state = \text{InProgress}$ at all times before $t_1$. Hence, by the definition of freezing (see Figure 8), $r$ is frozen for $U$ at all times between $t_0$ and $t_1$. □

**Corollary 29.** If a frozen step belongs to an SCX-record $U$, then each $r$ in $U.V$ is frozen for $U$ at all times between the first frozen step belonging to $U$ and the first commit step belonging to $U$.

**Proof.** Suppose there is a frozen step belonging to $U$. By Lemma 28, each $r$ in $U.V$ is frozen for $U$ at all times between the first freezing CAS belonging to $U$ on $r$ and the first commit step belonging to $U$. It follows directly from the pseudocode of Help that the first frozen step belonging to $U$ must follow the first freezing CAS belonging to $U$ on $r$, for each $r$ in $U.V$, and precede the first commit step belonging to $U$. □

**Corollary 30.** A successful mark step belonging to $U$ can occur only while $r$ is frozen for $U$.

**Proof.** Immediate from Lemma 27 and Corollary 29. □

**Lemma 31.** A Data-record can only be changed from unfrozen to frozen by a change in its info field (which can only be the result of a freezing CAS).

**Proof.** Let $r$ be a Data-record whose info field points to an SCX-record $U$. According to the definition of a frozen Data-record (see Figure 8), if $r.info$ does not change, then $r$ can only become frozen if $U.state$ changes from Committed or Aborted to InProgress, or from Aborted to Committed (provided $r$ is marked). However, both cases are impossible by Corollary 23. □

**Definition 32.** A Data-record $r$ is called permafrozen for SCX-record $U$ if $r$ is marked, $r.info$ points to $U$ and the $U.state$ is Committed. Notice that a Data-record that is permafrozen for $U$ is also frozen for $U$.

**Lemma 33.** Once a Data-record $r$ is permafrozen for SCX-record $U$, it remains permafrozen for $U$ thereafter.

**Proof.** By definition, when $r$ is permafrozen for $U$, it is frozen for $U$, $U.state$ is Committed and $r.marked = \text{True}$. Once $r.marked$ is set to True, it can never be changed back to False. By Corollary 23, $U.state$ was never Aborted, $U.state$ will remain Committed forever, and $r$ will be frozen for $U$ as long as $r.info$ points to $U$. It remains only to prove that $r.info$ cannot change while $r$ is permafrozen for $U$. Note that $r.info$ can be changed only by a successful freezing CAS.

<table>
<thead>
<tr>
<th></th>
<th>r.info.state</th>
<th>r.marked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frozen</td>
<td>Committed</td>
<td>True</td>
</tr>
<tr>
<td>Unfrozen</td>
<td>Committed</td>
<td>False</td>
</tr>
</tbody>
</table>
To obtain a contradiction, suppose a freezing CAS \( \text{fcas} \) changes \( r.info \) from \( U \) to \( W \) while \( r \) is permafrozen for \( U \). By Lemma 11, \( W \) is not the dummy SCX-record. Let \( S \) be the invocation of SCX that created \( W \). From the code of \text{Help}, \( r \) is in \( W.V \). So, by the precondition of SCX, there is an invocation of LLX\((r)\) linked to \( S \). By Observation 8.3 and line 25, the old value for \( \text{fcas} \) (a pointer to \( U \)) was read at line 4 of the LLX\((r)\) linked to \( S \). Let \( r \) be the invocation of LLX\((r)\) linked to \( S \). Since we have argued that \( U.state \) is never Aborted, \( U.state \in \{ \text{InProgress}, \text{Committed} \} \) when \( I \) reads \( state \) from \( U.state \) at line 5.

If \( state = \text{InProgress} \) then \( I \) does not enter the if-block at line 7, and returns \text{FAIL} or \text{FINALIZED}, which contradicts Definition 7.1.

Now, consider the case where \( state = \text{Committed} \). If we can argue that \( r \) is marked when \( I \) performs line 6, then we shall obtain the same contradiction as in the previous case. Since \( state = \text{Committed} \), a commit step belonging to \( U \) occurs before \( I \) performs line 5. By Lemma 27, any successful mark step belonging to \( U \) occurs prior to this commit step. Therefore, if \( r \) is \text{in.U.R}, then \( r \) will be marked when \( I \) performs line 6, and we obtain the same contradiction. The only remaining possibility is that \( r \) is not \text{in.U.R}, and \( r \) is marked by a successful mark step \text{mstep} belonging to some other SCX-record \( U' \) after \( I \) performs line 6, and before \( \text{fcas} \) occurs (which is while \( r \) is permafrozen for \( U \)). Since \( r.info \) points to \( U \) when \( I \) performs line 4, and again when \( \text{fcas} \) occurs, Lemma 12 implies that \( r.info \) points to \( U \) throughout this time. However, this contradicts Corollary 30, which states that \text{mstep} can only occur while \( r.info \) points to \( U' \).

**Lemma 34.** Suppose a successful mark step \text{mstep} belonging to an SCX-record \( U \) on \( r \) occurs. Then, \( r \) is frozen for \( U \) when \text{mstep} occurs, and forever thereafter.

**Proof.** By Corollary 30, \text{mstep} must occur while \( r \) is frozen for \( U \). From the code of \text{Help}, a frozen step belonging to \( U \) must precede \text{mstep}, and \( r \) must be in \( V \) (since it is marked at line 38). Thus, Corollary 29 implies that \( r \) is frozen for \( U \) at all times between \text{mstep} and the first commit step belonging to \( U \). Since \( r.marked \) is never changed from \text{True} to \text{False}, \text{mstep} must be the first mark step that ever modifies \( r.marked \). From the code of \text{Help}, \text{mstep} must precede the first commit step belonging to \( U \). If any commit step belonging to \( U \) occurs after \text{mstep}, immediately after the first such commit step, \( r \) will be marked, and \( r.info.state \) will be Committed, so \( r \) will become permafrozen for \( U \). By Lemma 33, \( r \) will remain frozen for \( U \), thereafter.

**Lemma 35.** Suppose \( I \) is an invocation of LLX\((r)\) that returns a value different from \text{FAIL} or \text{FINALIZED}. Then, \( r \) is not frozen at any time in \([t_0, t_1]\), where \( t_0 \) is when \( I \) reads \( r.info.state \) at line 5, and \( t_1 \) is when \( I \) reads \( r.info \) at line 9.

**Proof.** We prove that \( r \) is not frozen at any time between \( t_0 \) and \( t_1 \). Since \( I \) returns a value different from \text{FAIL} or \text{FINALIZED}, it enters the if-block at line 7, and sees \( r.info = \) \( r.info \) at line 9. Therefore, it sees either \( state = \text{Committed} \) and \( r.marked = \text{False} \), or \( state = \text{Aborted} \) at line 7. In each case, Corollary 23 guarantees that \( r.info.state \) will never change again after time \( t_0 \). Thus, if \( state = \text{Aborted} \), then \( r \) is not frozen at any time between \( t_0 \) and \( t_1 \). Now, suppose \( state = \text{Committed} \). We prove that \( r.marked \) does not change between \( t_0 \) and \( t_1 \). A pointer to an SCX-record \( W \) is read from \( r.info \) and stored in the local variable \( r.info \) at line 4, before \( t_0 \). At line 9, \( r.info \) still contains a pointer to \( W \). By Lemma 12, \( r.info \) must not change between line 4 and line 9. Therefore, \( r.info \) points to \( W \) at all times between \( t_0 \) and \( t_1 \). By Corollary 30, a successful mark step can occur between \( t_0 \) and \( t_1 \) only if it belongs to \( W \). Since \( state = \text{Committed} \), a commit step belonging to \( W \) must have occurred before \( t_0 \). By Lemma 27, any successful mark step belonging to \( W \) must have occurred before \( t_0 \). Therefore, \( W.state = \text{Committed} \) and \( r.marked = \text{False} \) throughout \([t_0, t_1]\).

**Corollary 36.** Let \( S \) be an invocation of SCX, and \( r \) be any Data-record in the \( V \) sequence of \( S \). Then, \( r \) is not frozen at any time in \([t_0, t_1]\), where \( t_0 \) is when the LLX\((r)\) linked to \( S \) reads \( r.info.state \) at line 5, and \( t_1 \) is when the LLX\((r)\) linked to \( S \) reads \( r.info \) at line 9.

**Proof.** Immediate from Definition 7.1 and Lemma 35.

### A.5 Properties of update CAS steps

**Observation 37.** An immutable field of a Data-record cannot change from its initial value.

**Proof.** This observation follows from the facts that Data-records can only be changed by SCX and an invocation of SCX can only accept a pointer to a mutable field as its \text{fld} argument (to modify).

**Observation 38.** Each mutable field of a Data-record can be modified only by a successful update CAS.

**Observation 39.** Each update CAS belonging to an SCX-record \( U \) is of the form \text{CAS}(\text{U.fld, U.old, U.new}). \( U.fld \) and \( U.new \) contain the arguments \( \text{fld} \) and \( \text{new} \), respectively, that were passed to the invocation of SCX\((V, R, \text{fld}, \text{new})\) that created \( U \).

**Proof.** An update CAS occurs at line 39 in an invocation of \text{Help}(\text{scxPtr.fld}), where it operates on \text{scxPtr.fld}, using \text{scxPtr.old} as its old value, and \text{scxPtr.new} as its new value. The fields of \text{scxPtr} do not change after \text{scxPtr} is created at line 21. At this line, the arguments \( \text{fld} \) and \( \text{new} \) that were passed to the invocation of SCX\((V, R, \text{fld}, \text{new})\) are stored in \text{scxPtr.fld} and \text{scxPtr.new}, respectively.

**Lemma 40.** The first update CAS belonging to an SCX-record \( U \) on a Data-record \( r \) occurs while \( r \) is frozen for \( U \).

**Proof.** Let \text{upcas} be the first update CAS belonging to \( U \). By line 39, such an update CAS will modify \text{U.fld} which, by Observation 8.1, is a mutable field of a Data-record \( r \) in \( U.V \). Since \text{upcas} is preceded by a frozen step in the pseudocode of \text{Help}, a frozen step belonging to \( U \) must precede \text{upcas}. Hence, Corollary 29 applies, and each \( r \) in \( U.V \) is frozen for \( U \) at all times between the first frozen step belonging to \( U \) and the first commit step \text{cstep} belonging to \( U \). From the code of \text{Help}, if \text{cstep} exists, then it must occur after \text{upcas}. Thus, when \text{upcas} occurs, \( r \) is frozen for \( U \).

In Section 4 we described a constraint on the use of SCX that allows us to implement an optimized version of SCX (which avoids the creation of a new Data-record to hold each value written to a mutable field), and noted that the
correctness of the unoptimized version follows trivially from the correctness of the optimized version. In order to prove the next few lemmas, we must invoke this constraint. In fact, we assume a weaker constraint, and are still able to prove what we would like to. We now give this weaker constraint, and remark that it is automatically satisfied if the constraint in Section 4 is satisfied.

**Constraint 41.** Let \( fild \) be a mutable field of a Data-record \( r \). If an invocation \( S \) of SCX(V, R, fld, new) is linearized, then:

- new is not the initial value of fild, and
- no invocation of SCX(V', R', fld, new) is linearized before the LLX(r) linked to S is linearized.

We prove the following six lemmas solely to prove that only the first update CAS belonging to an SCX-record can succeed. This result is eventually used to prove that exactly one successful update CAS belongs to any SCX-record which is helped to successful completion.

We need to know about the linearization of SCXs and linked LLXs to prove the next lemma, which uses Constraint 41. Let \( S \) be an invocation of SCX, and \( U \) be the SCX-record that it creates. As we shall see in Section A.8, we linearize \( S \) if and only if there is an update CAS belonging to \( U \), and \( S \) is linearized at its first update CAS. Each invocation of LLX linked to an invocation of SCX is linearized at line 9.

**Lemma 42.** No two update CASs belonging to different SCX-records can attempt to change the same field to the same value.

**Proof.** Suppose, to derive a contradiction, that update CASs belonging to two different SCX-records \( U \) and \( U' \) attempt to change the same (mutable) field of some Data-record \( r \) to the same value. Let upcas and upcas' be the first update CAS belonging to \( U \) and \( U' \), respectively. Let \( S \) and \( S' \) be the invocation of SCX that created \( U \) and \( U' \), respectively. From Observation 39 and the fact that upcas and upcas' attempt to change the same field to the same value, we know that \( S \) and \( S' \) must have been passed the same fild and new arguments. Note that \( S \) and \( S' \) are linearized at upcas and upcas', respectively. Without loss of generality, suppose \( S \) is linearized after \( S' \). By Constraint 41, \( S' \) is linearized after the invocation I of LLX(r) linked to \( S \) is linearized.

By Lemma 36, \( r \) is not frozen when \( I \) is linearized. By Lemma 40, \( r \) is frozen for \( U' \) when upcas' occurs (which is after \( I \) is linearized). By Lemma 31, \( r \) can become frozen for \( U' \) only by a successful freezing CAS belonging to \( U' \) on \( r \). Therefore, a successful freezing CAS feas' belonging to \( U' \) on \( r \) occurs after \( I \) is linearized, and before upcas'. By Lemma 40, \( r \) is frozen for \( U \) when upcas occurs (which is after upcas'), which implies that a successful freezing CAS feas belonging to \( U \) on \( r \) occurs after upcas', and before upcas. To recap, \( I \) is linearized before feas', which is before upcas', which is before feas, which is before upcas. By Lemma 25 and Observation 8.3, the old value old for feas is read from r.info and stored in r.info at line 4 by \( I \). Since \( I \) performs line 4 before it is linearized, old is read from r.info before feas'. Since feas' changes r.info to point to \( U' \), Lemma 12 implies that r.info does not point to \( U' \) at any time before feas'. Therefore, old is not \( U' \). Since feas is successful, r.info must be changed to old at some point after feas', and before upcas. However, this contradicts Lemma 12, since r.info had already contained old before feas'.

**Lemma 43.** An update CAS never changes a field back to its initial value.

**Proof.** By Observation 39, each update CAS belonging to an SCX-record \( U \) attempts to change a field to the value new that was passed as an argument to the invocation of SCX that created \( U \). Since Constraint 41 implies that new cannot be the initial value of the field, we know that no update CAS can change the field to its initial value.

**Lemma 44.** No update CAS has equal old and new values.

**Proof.** Let upcas be an update CAS and let \( U \) be the SCX-record to which it belongs. By Observation 39, the old value used by upcas is \( U.\text{old} \), and the new value used by upcas is \( U.\text{new} \). Let \( f \) be the field of a Data-record pointed to by \( U.\text{fild} \); this is the field to which upcas is applied. By Lemma 43, \( U.\text{new} \) cannot be the initial value of \( f \). If \( U.\text{old} \) is the initial value of \( f \), then we are done. So, suppose \( U.\text{old} \) is not the initial value of \( f \). Since a mutable field can only be changed by a successful update CAS, there exists a successful update CAS upcas' which changed \( f \) to \( U.\text{old} \) prior to upcas. By Observation 8.2, \( U.\text{old} \) was read from \( f \) prior to the start of the invocation \( S \) of SCX that created \( U \) and, therefore, prior to upcas. Hence, when upcas' occurs, \( U \) has not yet been created. Note that upcas' must occur in an invocation of HELP(ptr’) where ptr' points to some SCX-record \( U’ \) different from \( U \). However, by Lemma 42, upcas and upcas' use different new values, so \( U.\text{old} \) (the new value for upcas') must be different from \( U.\text{new} \) (the new value for upcas).

**Lemma 45.** At most one successful update CAS can belong to an SCX-record.

**Proof.** We prove this lemma by contradiction. Consider the earliest point in the execution when the lemma is violated. Let upcas' be the earliest occurring second successful update CAS belonging to any SCX-record, and \( U \) be the SCX-record to which it belongs, and let upcas_u be the preceding successful update CAS belonging to \( U \). Further, let \( f \) be the field upon which upcas_u operates, and let old and new be the old and new values used by upcas_u, respectively. By Observation 39, upcas_u and upcas_u' attempt to change the same field from the same old value to the same new value.) By Lemma 44, we know that old \( \neq \) new. Then, since upcas_u is successful, there must be a successful update CAS upcas_w belonging to some SCX-record \( W \) which changes \( f \) to old between upcas_u and upcas'_u. By Lemma 43, old is not the initial value of \( f \). Hence, there must be another successful update CAS upcas_w which changes \( f \) to old before upcas_u. By Lemma 42, upcas_w must belong to \( W \), so upcas_w and upcas'_u both precede upcas'_u. This contradicts the definition of upcas'_u.

**Lemma 46.** An update CAS never changes a field to a value that has already appeared there. (Hence, there is no ABA problem on mutable fields.)

**Proof.** Suppose a successful update CAS upcas belonging to an SCX-record \( U \) changes a field \( f \) to have value
new. By Lemma 43, new is not the initial value of f. By Lemma 45, all successful update CASs that change f must belong to different SCX-records. Hence, Lemma 42 implies that no update CAS other than upcas can change f to new.

Lemma 45 proved that at most one update CAS of each SCX-record can succeed. Now we prove that such a successful update CAS must be the first one belonging to SCX-record.

**Lemma 47.** Only the first update CAS belonging to an SCX-record U can succeed.

**Proof.** Let upcas be the first update CAS belonging to U, and f be the field that upcas attempts to modify. If upcas succeeds then, by Lemma 45, there can be no other successful update CAS belonging to U. So, suppose upcas fails. By Observation 39, each update CAS belonging to U uses the same old value U.old. By Observation 8.2, U.old was read from f prior to the start of the invocation S of SCX that created U and, therefore, prior to upcas. Then, since upcas fails, f must change between when U.old is read from f and when upcas occurs. By Observation 38, f can only be changed by an update CAS. By Lemma 46, each update CAS applied to f changes it to a value that it has not previously contained. Therefore, f will never again be changed to U.old. Hence, every subsequent update CAS belonging to U will fail.

A.6 Freezing works

In addition to being used to prove the remaining lemmas of this section, the following two results are used to prove linearizability in Section A.8. Intuitively, they allow us to determine whether a Data-record has changed simply by looking at its info field, and whether it is frozen.

**Corollary 48.** An update CAS belonging to an SCX-record U on a Data-record r can succeed only while r is frozen for U.

**Proof.** Suppose a successful update CAS upcas belongs to an SCX-record U. By Lemma 47, it is the first update CAS belonging to U. Lemma 40 proves the claim.

By Observation 38, a mutable field of r can only change while r is frozen.

**Lemma 49.** If a Data-record r is not frozen at time t₀, r.info points to an SCX-record U at or before time t₀, and r.info points to U at time t₁ > t₀, then no field of r is changed during [t₀, t₁).

**Proof.** Since r.info points to U at or before time t₀, and again at time t₁, Lemma 12 implies that r.info must point to U at all times in [t₀, t₁]. Further, from Lemma 31, r can only be changed from unfrozen to frozen by a change to r.info. Therefore, at all times in [t₀, t₁], r is not frozen. By Corollary 48, and Observation 38, each mutable field of r can change only while r is frozen. By Corollary 30, r.marked can change only while r is frozen. Finally, by Observation 37, immutable fields do not ever change. Hence, no field of r changes during [t₀, t₁].

The remaining results of this section describe intervals over which certain fields of a Data-record do not change. Suppose U is an SCX-record created by an invocation S of SCX, r is a Data-record in U.V, and I is the invocation of LLX(r) linked to S. Intuitively, we use the preceding lemma to prove, over the next two lemmas, that no field of r changes between when I last reads r.info, and when r is frozen for U. We then use this result in Section A.7 to prove that S succeeds if and only if this holds for each r in V. The remaining results of this section are used primarily to prove that exactly one successful update CAS belongs to U if a frozen step belongs to U (and S does not crash, or some process helps it complete).

**Corollary 50.** Let U be an SCX-record, and S be the invocation of SCX that created U. If there is a successful freezing CAS belonging to U on r, then no field of r changes after the LLX(r) linked to S reads r.info.state at line 5, and before this freezing CAS occurs.

**Proof.** Let fcas be a successful freezing CAS belonging to U on r. Note that the LLX(r) linked to S terminates before S begins. Since S creates U and fcas changes r.info to point to U, S begins before fcas. We now check that Lemma 49 applies. By Corollary 36, r is not frozen when the LLX(r) linked to S executes line 5. From line 25 of HELP and Observation 8.3, we know the old value r.info for fcas is read from r.info at line 4 by the LLX(r) linked to S. Further, since fcas succeeds, r.info contains r.info just prior to fcas. Thus, Lemma 49 applies, and proves the claim.

**Lemma 51.** If an update CAS belongs to an SCX-record U then, for each r in U.V, there is a successful freezing CAS belonging to U on r, and no mutable field of r changes during [t₀(r), t₁), where t₀(r) is when the first such freezing CAS occurs, and t₁ is when the first update CAS belonging to U occurs.

**Proof.** Suppose an update CAS belongs to an SCX-record U. Let upcas be the first such update CAS. Since each update CAS is preceded in the code by a frozen step, a frozen step also belongs to U. Fix any r in U.V. By Lemma 21, there is a successful freezing CAS belonging to U on r. By Lemma 28, r is frozen for U at all times in [t₀(r), t₂), where t₂ is when the first commit step belonging to U occurs. Since an update CAS belonging to an SCX-record W can modify r only while r is frozen for W (by Corollary 48), any update CAS that modifies r during [t₀(r), t₂) must belong to U. From the code of HELP, t₀(r) < t₁ < t₂. However, since the first update CAS belonging to U occurs at t₁, no update CAS belonging to U can occur during [t₀(r), t₁).

**Lemma 52.** Let U be an SCX-record created by an invocation S of SCX, and r be a Data-record in U.V. Let t₀ be when the LLX(r) linked to S reads U.state at line 5, and t₂ be when the first commit step belonging to U occurs. If an update CAS belongs to U then, for each r in U.V., between t₀ and t₂, r.marked can be changed (from False to True, by the first mark step belonging to U on r) only if r is in U.R.

**Proof.** Fix any r in U.V. The fact that r.marked can be changed only from False to True follows immediately from the fact that r.marked is initially False, and is only changed at line 38. It also follows that a successful mark step on r must be the first mark step on r. The rest of the claim is more subtle. Suppose a successful mark step mstep belonging to an SCX-record W on r occurs during (t₀, t₂). By
Lemma 34. \( r \) is frozen for \( W \) when \( mstep \) occurs. From the code of \textsc{Help}, a frozen step \( fstep \) belonging to \( U \) precedes the first update CAS belonging to \( U \), and a freezing CAS belonging to \( U \) on \( r \) precedes \( fstep \). By Lemma 17, there is a successful freezing CAS belonging to \( U \) on \( r \). By Lemma 14, it must be the first freezing CAS \( feas \) belonging to \( U \) on \( r \). Let \( t_1 \) be when \( feas \) occurs. Note that \( t_0 < t_1 < t_2 \). By Corollary 50, \( r \) does not change during \([t_0, t_1]\). Thus, \( mstep \) cannot occur in \([t_0, t_1]\). By Lemma 28, \( r \) is frozen for \( U \) at all times during \([t_1, t_2]\). This implies \( U = W \), so \( mstep \) is the first mark step belonging to \( U \) on \( r \). Finally, since there is a mark step belonging to \( U \) on \( r \), we obtain from line 38 that \( r \) is in \( U.R \). \( \square \)

A.7 Correctness of \textsc{Help}

The following lemma shows that a helper of an SCX-record cannot return \texttt{True} until after the SCX-record is Committed. We shall use this to ensure that the SCX does not return \\texttt{True} until after the SCX has taken effect.

Lemma 53. An invocation of \textsc{Help}(\texttt{scxPtr}) where \texttt{scxPtr} points to an SCX-record \( U \) cannot return from line 31 before the first commit step belonging to \( U \).

Proof. Suppose an invocation \( H \) of \textsc{Help}(\texttt{scxPtr}) returns at line 31. Before returning, \( H \) sees \( r.info \neq \texttt{scxPtr} \) at line 27, which implies that \( r.info \) does not point to \( U \). Prior to this, \( H \) performs a freezing CAS belonging to \( U \) at line 26. By line 29, a frozen step belongs to \( U \). Then, since Lemma 28 states that \( r.info \) points to \( U \) at all times between the first freezing CAS belonging to \( U \) and the first commit step belonging to \( U \), a commit step belonging to \( U \) must occur before \( H \) returns. \( \square \)

Next, we obtain an exact characterization of the update CAS steps that succeed.

Lemma 54. If there is an update CAS belonging to an SCX-record \( U \), then the first update CAS belonging to \( U \) is successful and changes the mutable field pointed to by \( U.fld \) from \( U.old \) to \( U.new \). No other update CAS belonging to \( U \) is successful.

Proof. Let \( t_0(r) \) be when the LLX(\( r \)) linked to \( S \) reads \( r.info.state \) at line 5, and \( t_1 \) be when the first update CAS belonging to \( U \) occurs. Since an update CAS belongs to \( U \), Lemma 51 implies that, for each \( r \) in \( U.V \), no mutable field of \( r \) changes between \( t_0(r) \) and \( t_1 \). From Observation 8.2 and the code of LLX, we see that the value stored in \( U.old \) is read from the field pointed to by \( U.fld \) after time \( t_0(r) \). Further, since the LLX(\( r \)) linked to \( S \) terminates before \( S \) begins (by the definition of an LLX linked to an SCX) and, in turn, before \( U \) is created, we know the value stored in \( U.old \) was read before any update CAS belonging to \( U \) occurred. Then, since \( U.fld \) is a mutable field of a Data-record in \( U.V \), this field does not change between \( t_0(r) \) and \( t_1 \). By Observation 39, any update CAS belonging to \( U \) will attempt to change \( U.fld \) from \( U.old \) to \( U.new \), so the first update CAS belonging to \( U \) will succeed. Lemma 47 completes the proof. \( \square \)

Our next lemma shows that the SCXs that are linearized have the desired effect.

Lemma 55. Let \( U \) be an SCX-record created by an invocation \( S \) of SCX, and \texttt{ptr} point to \( U \). If either

- a frozen step belongs to \( U \) and some invocation of \textsc{Help}(\texttt{ptr}) terminates, or
- \( S \) or any invocation of \textsc{Help}(\texttt{ptr}) returns \texttt{True},

then the following claims hold.

1. Every invocation of \textsc{Help}(\texttt{ptr}) that terminates returns \texttt{True}.
2. Exactly one successful update CAS belongs to \( U \), and it is the first update CAS belonging to \( U \). It changes the mutable field pointed to by \( U.fld \) from \( U.old \) to \( U.new \).
3. A frozen step of \( U \) and a commit step of \( U \) occur before any invocation of \textsc{Help}(\texttt{ptr}) returns.
4. At all times after the first commit step for \( U \), each \( r \) in \( U.R \) is permafrozen for \( U \).

Proof. We first simplify the lemma’s hypothesis. If \( S \) returns \texttt{True}, then \( S \)’s invocation of \textsc{Help}(\texttt{ptr}) has returned \texttt{True}. If some invocation of \textsc{Help}(\texttt{ptr}) returns \texttt{True}, a commit step belongs to \( U \) by Lemma 53, and that commit step is preceded by a frozen step of \( U \). So, for the remainder of the proof, we can assume that a frozen step belongs to \( U \) and some invocation of \textsc{Help}(\texttt{ptr}) terminates.

Proof of Claim 1: Since a frozen step belongs to \( U \), Lemma 21 implies that no abort step belongs to \( U \). Thus, \( H \) cannot return at line 35, which implies that \( H \) must return \texttt{True}.

Proof of Claim 2 and Claim 3: By Claim 1, \( H \) must return \texttt{True}. If \( H \) returns at line 42, then it does so after performing an update CAS and a commit step, each belonging to \( U \). Otherwise, \( H \) returns at line 31. However, by Lemma 53, no invocation of \textsc{Help}(\texttt{ptr}) can return at line 31 until the first commit step belonging to \( U \), which is necessarily preceded by an update CAS for \( U \) (by inspection of \textsc{Help}). Thus, Claim 3 is proved. Lemma 54 proves Claim 2.

Proof of Claim 4: By Corollary 29, every \( r \) in \( U.V \) is frozen for \( U \) from the first frozen step belonging to \( U \) until the first commit step belonging to \( U \). From the code of \textsc{Help}, each \( r \) in \( U.R \) is marked before the first commit step belonging to \( U \), and Lemma 52 implies that they are still marked when the first commit step belonging to \( U \) occurs. Further, immediately after the first commit step belonging to \( U \) (which must exist by Claim 3), \( U.state \) will be Committed, so each \( r \) that is in both \( U.V \) and \( U.R \) will be permafrozen for \( U \). Since \( R \) is a subsequence of \( V \), and \( U.R \) and \( U.V \) do not change after they are obtained at line 21 from \( R \) and \( V \), respectively, it follows from Lemma 33 that each \( r \) in \( U.R \) remains permafrozen for \( U \) forever. \( \square \)

Now we show that SCXs that are not linearized do not modify any mutable fields, and do not return \texttt{True}.

Lemma 56. Let \( U \) be an SCX-record created by an invocation \( S \) of SCX, and \texttt{ptr} be a pointer to \( U \). If \( S \) or any invocation \( H \) of \textsc{Help}(\texttt{ptr}) returns \texttt{False}, then the following claims hold.

1. Every invocation of \textsc{Help}(\texttt{ptr}) that terminates returns \texttt{False}.
2. An abort step belonging to \( U \) occurs before any invocation of \textsc{Help}(\texttt{ptr}) returns.
3. No update CAS belongs to \( U \).
Proof. Note that, if $S$ returns False, then its invocation of $\text{Help}(ptr)$ returns False, so an invocation $H$ exists and returns False. By Lemma 55.1, if any invocation of $\text{Help}(ptr)$ returned True, then $H$ would have to return True. Since $H$ returns False, every terminating invocation of $\text{Help}(ptr)$ must return False, which proves Claim 1. We now prove Claim 2 and Claim 3. Consider the invocation $H'$ of $\text{Help}(ptr)$ that returns earliest. By Claim 1, $H'$ returns False. Before $H'$ returns False, an abort step belonging to $U$ is performed at line 34. Thus, Lemma 21 implies that no frozen step belongs to $U$. By the code of $\text{Help}$, each update CAS belonging to $U$ follows a frozen step belonging to $U$. □

Now that we have proved each invocation of $\text{Help}$ that returns True or False has its expected effect, we must prove that the return value is always correct. (Otherwise, for example, two invocations of SCX with overlapping $V$ sequences could interfere with one another, but still return True.)

Lemma 57. Let $U$ be an SCX-record created by an invocation $S$ of SCX, and $ptr$ be a pointer to $U$. Any invocation $H$ of $\text{Help}(ptr)$ that terminates returns True if no $r$ in $U.V$ changes from when the LLX($r$) linked to $S$ reads $r$’s info at line 9 at time $t_0(r)$ to when the first freezing CAS belonging to $U$ on $r$ at time $t_1(r)$. Otherwise, $H$ returns False.

Proof. Since $H$ terminates, it must return True or False. Hence, it suffices to prove $H$ returns True if and only if no $r$ in $U.V$ changes between $t_0(r)$ and $t_1(r)$.

Case I: Suppose $H$ returns True. By Lemma 55.3, a frozen step belongs to $U$. Hence, by Lemma 17, there is a successful freezing CAS belonging to $U$ for each $r$ in $U.V$. Then, by Corollary 50, no field of $r$ changes between $t_0(r)$ and $t_1(r)$.

Case II: Suppose $H$ returns False. We show that some $r$ in $U.V$ changes between $t_0(r)$ and $t_1(r)$. Since $H$ can only return False at line 35, immediately before $H$ returns, it performs an abort step belonging to $U$. Thus, Lemma 24.4 applies and the claim is proved. □

A.8 Linearizability of LLX/SCX/VLX

As described in Section 3, we linearize all reads, all invocations of LLX that do not return FAIL, all invocations of VLX that return TRUE, and all invocations of SCX that modify the sequential data structure (all that return TRUE, and some that do not terminate). In our implementation, subtle interactions with a concurrent invocation of SCX can cause an invocation of LLX to return FAIL. Since this cannot occur in a linearized execution, we do not linearize any such invocation of LLX. Similarly, we allow some invocations of SCX and VLX to return FALSE because of interactions with concurrent invocations of SCX. Rather than distinguishing between the invocations of SCX or VLX that return FALSE because of an earlier linearized invocation of SCX (which is allowed by the sequential specification), and those that return FALSE because of contention, we simply opt not to linearize any invocation of SCX or VLX that returns FALSE. Alternatively, we could have accounted for the invocations that returns FALSE because of contention by allowing spurious failures in the sequential specification of the operations. However, this would unnecessarily complicate the sequential specification. Intuitively, an algorithm designer using LLX, SCX and VLX is most likely to be interested in invocations of SCX and VLX that return TRUE since these, respectively, change the sequential data structure, and indicate that a set of Data-record have not changed since they were last passed to successful invocations of LLX by this process. Knowing whether an invocation of SCX or VLX was unsuccessful because of a change to the sequential data structure, or because of contention, is less likely to be useful.

Before we give the linearization points, we state precisely which invocations of SCX we shall linearize. Let $S$ be an invocation of SCX, and $U$ be the SCX-record it creates. We linearize $S$ if and only if there is an update CAS belonging to $U$. By Lemma 55, every successful invocation of SCX will be linearized. (By Lemma 56, no unsuccessful invocation of SCX will be linearized.)

We first give the linearization points of the operations, then we prove our LLX/SCX/VLX implementation respects the correctness specification given in Section 3.

Linearization points:

• An LLX($r$) that returns values at line 11 is linearized at line 9. We linearize an LLX($r$) that returns Finalized at line 13.

• Let $U$ be an SCX-record created by an invocation $S$ of SCX. Suppose there is an update CAS belonging to $U$. We linearize $S$ at the first such update CAS (which is the unique successful update CAS belonging to $U$, by Lemma 54).

• An invocation $I$ of VLX that returns True is linearized at the first execution of line 47.

• We assume reads are atomic. Hence, a read is simply linearized when it occurs.

Lemma 58. The linearization point of each linearized operation occurs during the operation.

Proof. This is trivial to see for reads, and invocations of LLX and VLX. Let $U$ be an SCX-record created by a linearized invocation $S$ of SCX, and $ptr$ be a pointer to $U$. This claim is not immediately obvious for $S$, since the first update CAS belonging to $U$ may be performed by a process helping $U$ to complete (not the process performing $S$). Since $S$ is linearized, there is an update CAS $upcas$ belonging to $U$, which can only occur in an invocation of $\text{Help}(ptr)$. Since $ptr$ points to $U$, which is created by $S$, $upcas$ must occur after the start of $S$. If $S$ does not terminate, then we are done. Otherwise, Lemma 55.3 implies that a commit step belongs to $U$, and the first such commit step occurs before any invocation of $\text{Help}(ptr)$ returns. From the code of $\text{Help}$, $upcas$ must occur before the first commit step belonging to $U$, so $upcas$ must occur before any invocation of $\text{Help}(ptr)$ returns. Finally, since $S$ invokes $\text{Help}(ptr)$, $upcas$ must occur before $S$ terminates. □

We first show that each read returns the correct result according to its linearization point.

Lemma 59. If a read $R_f$ of a field $f$ is linearized after a successful invocation of SCX$(V, R, fld, new)$, where $fld$ points to $f$, then $R_f$ returns the parameter new of the last such SCX. Otherwise, $R_f$ returns the initial value of $f$. 
Proof. We proceed by cases.

Case I: \( f \) is an immutable field. In this case, a pointer to \( f \) cannot be the \( fld \) parameter of an invocation of \( SCX \). Further, by Observation 37, \( f \) cannot be modified after its initialization, so \( R_f \) returns the initial value of \( f \).

Case II: \( f \) is a mutable field. Suppose there is no successful invocation of \( SCX(V, R, fld, new) \), where \( fld \) points to \( f \), linearized before \( R_f \). Since an invocation of \( SCX \) is linearized at its first update CAS, there can be no update CAS on \( f \) prior to \( R_f \). Since \( f \) can only be modified by successful update CAS, \( R_f \) must return the initial value of \( f \).

Now, suppose there is a successful invocation of \( SCX(V, R, fld, new) \), where \( fld \) points to \( f \), linearized before \( R_f \). Let \( S \) be the last such invocation of \( SCX \) linearized before \( R_f \), and \( U \) be the \( SCX \)-record it creates. By Lemma 55.2, there is exactly one successful update CAS \( upcas \) belonging to \( U \), occurring at the linearization point of \( S \). Since each successful update CAS is the linearization point of some invocation of \( SCX \), and no invocation of \( SCX(V', R', fld, new') \) is linearized between \( S \) and \( R_f \), no successful update CAS occurs between \( S \) and \( R_f \). Since a mutable field can only be changed by update CAS, \( R_f \) returns the value stored by the successful update CAS \( upcas \) belonging to \( U \). By Lemma 55.2, \( upcas \) changes \( f \) to \( U.new \). Finally, since \( U.new \) does not change after it is obtained from \( new \) at line 21, the claim is proved.

Next, we prove that an LLX that returns a snapshot does return the correct result according to its linearization point.

Corollary 60. Let \( r \) be a Data-record with mutable fields \( f_1, \ldots, f_n \), and \( I \) be an invocation of LLX(\( r \)) that returns a tuple of values \( (m_1, \ldots, m_n) \) at line 11. For each mutable field \( f_i \) of \( r \), if \( I \) is linearized after a successful invocation of \( SCX(V, R, fld, new) \), where \( fld \) points to \( f_i \), then \( m_i \) is the parameter new of the last such invocation of \( SCX \). Otherwise, \( m_i \) is the initial value of \( f_i \).

Proof. Since \( I \) returns at line 11, the same value is read from \( r.info \) on line 4 and line 9 at times \( t_0 \) and \( t_1 \), respectively. By Lemma 35, \( r \) is unfrozen at line 7, which is between \( t_0 \) and \( t_1 \). Thus, Lemma 49 implies that \( r \) does not change during \( [t_0, t_1] \). Since the values returned by the LLX are read from the fields of \( r \) between \( t_0 \) and \( t_1 \) (at line 8), each read returns the same result as it would if it were executed atomically at the linearization point of the LLX (line 9). Finally, Lemma 59 completes the proof.

Next, we show that an LLX(\( r \)) returns \( \text{Finalized} \) only if \( r \) really has been finalized.

Lemma 61. Let \( I \) be an invocation of LLX(\( r \)), and \( U \) be the \( SCX \)-record to which \( I \) reads a pointer at line 4. If \( I \) returns \( \text{Finalized} \), then an invocation \( S \) of \( SCX(V, R, fld, new) \) that created \( U \) is linearized before \( I \), and \( r \) is in \( R \).

Proof. Suppose \( I \) returns \( \text{Finalized} \). Then, \( I \) is linearized at line 13. When \( I \) performs line 12, either \( r.info.state \) is Committed or \( I \)'s invocation of Help(\( r.info \)) returns True. We show that, in either case, a commit step belonging to \( U \) must have occurred before \( I \) performs line 13. If \( r.info.state \) is Committed, then this follows immediately from the fact that no \( SCX \)-record has Committed as its initial state. Otherwise, Lemma 55.3 implies that a commit step belonging to \( U \) occurs before \( I \)'s invocation of Help(\( r.info \)) returns. Since a commit step belongs to \( U \), Lemma 11 implies that \( U \) is not the dummy \( SCX \)-record, so there must be an invocation \( S \) of \( SCX(V, R, fld, new) \) that created \( U \). From the code of Help, an update CAS belonging to \( U \) occurs before the first commit step belonging to \( U \). Since \( S \) is linearized at its first update CAS, \( S \) is linearized before \( I \).

It remains to show that \( r \) is in \( R \). Since \( U.R \) does not change after it is obtained from \( R \) at line 21, it suffices to show \( r \) is in \( U.R \). By line 12, \( marked_1 = \text{True} \), which means that \( r \) is marked when \( I \) reads a pointer to \( U \) from \( r.info \) at line 4. Let \( t_0 \) be when \( I \) performs line 4, and \( t_1 \) be when the first commit step belonging to \( U \) occurs. We consider two cases. Suppose \( t_1 < t_0 \). Then, when \( I \) performs line 4, \( r \) is marked, \( r.info \) points to \( U \), and \( U.state = \text{Committed} \), which means that \( r \) is permafrozen for \( U \). By Lemma 33, \( r \) is frozen for \( U \) at all times after \( t_0 \). Now, suppose \( t_0 < t_1 \). In this case, Lemma 27 implies \( U.state = \text{InProgress} \) when \( I \) performs line 4, and that \( U.state \) will never be Aborted. By Lemma 15, \( r.info \) must point to \( U \) at all times in \( [t_0, t_1] \). Thus, at \( t_1 \), \( r \) is marked and \( r.info \) points to \( U \), which means that \( r \) is permafrozen for \( U \). By Lemma 33, \( r \) is frozen for \( U \) at all times after \( t_1 \). In each case, \( r \) is frozen for \( U \) at all times in some (non-empty) suffix of the execution. Since \( r \) is marked, there must be a successful mark step belonging to some \( SCX \)-record \( W \) on \( r \). By Lemma 34, \( r \) is frozen for \( W \) at all times after this mark step. Therefore, \( U = W \), which means there is a mark step belonging to \( U \) on \( r \). Finally, line 38 implies that \( r \) is in \( U.R \).

Lemma 62. Let \( r \) be a Data-record, \( I \) be an invocation of LLX(\( r \)) that terminates, and \( S \) be a linearized invocation of \( SCX(V, R, fld, new) \) with \( r \) in \( R \). \( I \) returns \( \text{Finalized} \) if it is linearized after \( S \), or begins after \( S \) is linearized. (This implies \( I \) will be linearized in both cases.)

Proof. Let \( U \) be the \( SCX \)-record created by \( S \).

Case I: \( I \) begins after \( S \) is linearized. In this case, \( S \) is linearized at a successful update CAS \( upcas \) belonging to \( U \) that occurs before \( I \) begins. From the code of Help, a mark step on \( r \) belonging to \( U \) must occur before \( upcas \). Consider the first mark step \( mstep \) on \( r \). Since \( r.marked \) is initially \( \text{False} \), \( mstep \) must be successful. Let \( W \) be the \( SCX \)-record to which \( mstep \) belongs. By Lemma 34, \( r \) is frozen for \( W \) at all times after \( mstep \). By Lemma 48, \( r \) is frozen for \( U \) when \( upcas \) occurs (which is after \( mstep \)). Since \( r \) can be frozen for only one \( SCX \)-record at a time, \( W = U \). Therefore, \( r.info \) points to \( U \) throughout \( I \). So, when \( I \) performs line 7, it will see \( state = \text{Committed} \) and \( marked_1 = \text{True} \). Moreover, when it subsequently performs line 12, it will see \( r.info.state = \text{Committed} \) and \( marked_1 = \text{True} \), so it will return \( \text{Finalized} \).

Case II: \( I \) is linearized after \( S \). \( I \) can either be linearized at line 9 or at line 13. If it is linearized at line 13, then it returns \( \text{Finalized} \), and we are done. Suppose, in order to derive a contradiction, that \( I \) is linearized at line 9. Then, \( I \) returns at line 11 and, by Corollary 36, \( r \) is unfrozen at all times during \( [t_0, t_1] \), where \( t_0 \) is when \( I \) performs line 5, and \( t_1 \) is when \( I \) performs line 9. Since \( I \) is linearized at time \( t_1 \), \( S \) must be linearized at an update CAS \( upcas \) belonging to \( U \) that occurs before time \( t_1 \). By Corollary 48, \( upcas \) can only occur while \( r \) is frozen for \( U \). Since \( r \) is unfrozen at all times during \( [t_0, t_1] \), \( upcas \) must occur at some point before \( t_0 \). From the code of Help, a mark step belonging to \( U \) must occur before \( upcas \). Consider the first mark step
Let

\[ L \]

update CAS belonging to

\[ L \]

when it is supposed to, according to the correctness specification

\[ I \]

must contain

\[ I \]

linked to

\[ I \]

in the previous case,

\[ L \]

is unfrozen at all times after

\[ L \]

must contain

\[ L \]

linked to

\[ L \]

infinite). For an invocation \( I \) of \( VLX(V) \) or \( SCX(V, R, \text{fld, new}) \) is linearized then, for each \( r \in V \), no \( SCX(V', R', \text{fld}', \text{new}') \) with \( r \in V' \) is linearized between the \( LLX(r) \) linked to \( I \) and \( I \).

**Proof.** Fix any \( r \in V \). By the preconditions of SCX and VLX, there must be an \( LLX(r) \) linked to \( I \). If \( I \) is an invocation of SCX, then let \( U \) be the SCX-record that it creates. Let \( L \) be the \( LLX(r) \) linked to \( I \), \( t_0 \) be when \( L \) performs line 4, \( t_1 \) be when \( L \) performs line 5 and \( t_2 \) be when \( L \) is linearized (at line 9). Since \( L \) is linked to \( I \), \( t_2 \) exists. Let \( t_3 \) be when \( L \) is linearized. For \( SCX \), \( t_3 \) is when the first update CAS belonging to \( U \) occurs. For \( VLX \), \( t_3 \) is when \( I \) first performs line 47. Since \( I \) is linearized, \( t_3 \) exists. Finally, we define time \( t_4 \). For SCX, \( t_4 \) is the first commit step belonging to \( U \) occurs, or the end of the execution if there is no commit step belonging to \( U \) (or \( \infty \) if the execution is infinite). For \( VLX \), \( t_4 \) is if \( I \) sees \( r.info = r.info \) at line 47 (in the iteration for \( r \)). If \( I \) is an invocation of \( VLX \) then, since \( I \) is linearized, it returns True. This implies that \( I \) must see \( r.info = r.info \) at line 47 in the iteration for \( r \), so \( t_4 \) exists. Clearly, \( t_4 \) exists if \( I \) is an invocation of \( SCX \).

We now prove \( t_0 < t_1 < t_2 < t_3 < t_4 \). From the code of \( LLX \), \( t_0 < t_1 < t_2 \). Suppose \( I \) is an invocation of \( VLX \). Since \( L \) terminates before \( I \) begins, \( t_2 < t_3 \). Trivially, \( t_3 < t_4 \). Now, suppose \( I \) is an invocation of \( SCX \). Since each update CAS belonging to \( U \) occurs in an invocation of \( HELP(ptr) \) where \( ptr \) points to \( U \), and \( U \) is created during \( I \), \( t_3 \) must occur after the start of \( I \). Since \( L \) terminates before \( I \) begins, \( t_2 < t_3 \). From the code of \( HELP \), the first update CAS belonging to \( U \) precedes the first commit step belonging to \( U \) (as well as the other options for \( t_4 \)), so \( t_3 < t_4 \).

Next, we prove that, at all times in \( [t_1, t_4] \), \( r \) is either frozen for \( U \), or not frozen (i.e., \( r \) is not frozen for any SCX-record different from \( U \) at any point during \( [t_1, t_4] \)). We consider two cases.

**Case I:** Suppose \( I \) is an invocation of \( VLX \). Then, \( r.info \) contains \( r.info \) at \( t_0 \), and again at \( t_4 \). By Lemma 12, \( r.info \) must contain \( r.info \) at all times in \( [t_0, t_4] \). By Corollary 36, \( r \) is unfrozen at time \( t_1 \). By Lemma 31, \( r \) can only be changed by \( r.info \). Since \( r \) does not change during \( [t_0, t_4] \), \( r \) is unfrozen at all times in \( [t_1, t_4] \).

**Case II:** Suppose \( I \) is an invocation of \( SCX \). From the code of \( HELP \), a frozen step belonging to \( U \) precedes the first update CAS belonging to \( U \). By Lemma 17, a successful freezing CAS belonging to \( U \) on \( r \) precedes the first frozen step belonging to \( U \). Let \( t_2 \) be when the first successful freezing CAS \( fcas \) belonging to \( U \) on \( r \) occurs. It follows that \( t_2 < t_4 \). Since each freezing CAS belonging to \( U \) occurs in an invocation of \( HELP(ptr) \) where \( ptr \) points to \( U \), and \( U \) is created during \( I \), each freezing CAS belonging to \( U \) must occur after the start of \( I \). Recall that \( L \) terminates before the start of \( I \). Hence, \( t_2 > t_2 \). By Observation 8.3 and line 25, the old value for \( fcas \) is the value \( r.info \) that was read from \( r.info \) at line 4 of \( L \) (at \( t_0 \)). Since \( fcas \) is successful, \( r.info \) must contain \( r.info \) just before \( fcas \). Therefore, \( r.info \) contains \( r.info \) at \( t_0 \), and again at \( t_2 \). By the same argument we made in Case I (but with \( t_2 \) instead of \( t_4 \)), Lemma 12, Corollary 36, and Lemma 31 imply that \( r \) is unfrozen at all times in \( [t_1, t_2] \). By Lemma 28, \( r \) is frozen for \( U \) at all times in \( [t_2, t_4] \), which proves this case.

At last, we have assembled the results needed to obtain a contradiction. Suppose, to derive a contradiction, that an invocation \( S \) of \( SCX(V', R', \text{fld}', \text{new}') \) with \( r \in V' \) is linearized between \( L \) and \( I \) (i.e., \( (t_2, t_4) \)). Let \( W \) be the SCX-record created by \( S \). \( S \) is linearized at the first update \( upcas \) belonging to \( W \). From the code of \( HELP \), a frozen step belonging to \( W \) precedes \( upcas \), and \( upcas \) precedes any commit step belonging to \( W \). By Lemma 28, a successful freezing CAS \( fcas \) belonging to \( W \) on \( r \) precedes \( upcas \), and \( r \) is frozen for \( W \) at all times after \( fcas \), and before the first commit step belonging to \( W \). Therefore, \( r \) is frozen for \( W \) when \( upcas \) occurs in \( (t_2, t_4) \). Since, at all times in \( [t_1, t_4] \), \( r \) is either frozen for \( U \), or not frozen, we must have \( W = U \). This is a contradiction, since \( upcas \) occurs before the first update CAS belonging to \( U \) occurs (at \( t_4 \)). Thus, \( S \) cannot exist.

**Theorem 64.** Our implementation of \( LLX/SCX/VLX \) satisfies the correctness specification discussed in Section 3. That is, we linearize all successful \( LLX \)s, all successful \( SCX \)s, a subset of the \( SCX \)s that never terminate, all successful \( VLX \)s, and all reads, such that:

1. Each read of a field \( f \) of a Data-record \( r \) returns the last value stored in \( f \) by a linearized SCX (or \( f \)'s initial value, if no linearized SCX has modified \( f \)).
2. Each linearized \( LLX(r) \) that does not return \( FINALIZED \) returns the last value stored in each mutable field \( f \) of \( r \) by a linearized SCX (or \( f \)'s initial value, if no linearized SCX has modified \( f \)).
3. Each linearized \( LLX(r) \) returns \( FINALIZED \) if and only if it is linearized after an \( SCX(V, R, \text{fld, new}) \) with \( r \) in \( R \).
4. If an invocation \( I \) of \( SCX(V, R, \text{fld, new}) \) or \( VLX(V) \) returns True then, for all \( r \in V \), there has been no \( SCX(V', R', \text{fld}', \text{new}') \) with \( r \in V' \) linearized since the \( LLX(r) \) linked to \( I \).

**Proof.** By Lemma 58, the linearization point of each operation occurs during that operation. Claim 1 follows immediately from Lemma 59. Claim 2 is immediate from Lemma 60. The only-if direction of Claim 3 follows from Lemma 61, and the if direction follows from Lemma 62. Claim 4 is immediate from Lemma 63.

### A.9 Progress Guarantees

**Lemma 65.** \( LLX \), \( SCX \) and \( VLX \) are wait-free

**Proof.** The loop in \( H \) iterates over the elements of the finite sequence \( V \) and performs a constant amount of work during each iteration. If \( H \) does not return from the loop, then it performs a constant amount of work after the loop and returns. The claim then follows from the code. □
Lemma 66. Our implementation satisfies the first progress property in Section 3: Each terminating LLX(r) returns Finalized if it begins after the end of a successful SCX(V, R, fld, new) with r ∈ R or after another LLX(r) has returned Finalized.

Proof. Consider a terminating invocation I’ of LLX(r). If I’ begins after the end of a successful SCX(V, R, fld, new) with r ∈ R, the claim follows from Lemma 62. If I’ begins after another invocation I of LLX(r) has returned Finalized, then I is linearized after an invocation S of SCX(V, R, fld, new) with r ∈ R. Since I precedes I’, I’ starts after S is linearized. By Lemma 62, I’ returns Finalized. □

We now begin to prove the non-blocking progress properties. First, we design a way to assign blame to an SCX for each failed invocation of LLX, VLX or SCX.

Definition 67. Let I be an invocation of LLX that returns Fail. If I enters the if-block at line 7, then let U be the SCX-record pointed to by r.info when I performs line 9. Otherwise, let U be the SCX-record pointed to by r.info when I performs line 4. We say I blames the invocation S of SCX that created U. (We prove below that S exists.)

Lemma 68. If an invocation I of LLX returns Fail, then I blames some invocation of SCX.

Proof. Suppose I enters the if-block at line 7. Then, I must see r.info ≠ r.info at line 9. Let U be the SCX-record pointed to by r.info when I performs line 9. Since I reads r.info from r.info at line 4, r.info must change to point to U between when I performs line 4 and line 9. Thus, there must be a successful freezing CAS belonging to U on r between these two times. Since a successful freezing CAS belongs to U, Lemma 11 implies that U cannot be the dummy SCX-record. Therefore, U must be created by an invocation of SCX.

Now, suppose I does not enter the if-block at line 7. Then, from the code of LLX, r.info.state cannot be Aborted when I performs line 5, so the SCX-record pointed to by r.info is not the dummy SCX-record. Since I reads the value stored in r.info from r.info at line 4, the SCX-record pointed to by r.info when I performs line 4 must have been created by an invocation of SCX. □

Definition 69. Let I be an invocation of VLX(V) that returns False, and r be the Data-record in V for which I sees r.info ≠ r.info at line 47. Consider the first successful freezing CAS on r between when the LLX(r) linked to I reads r.info at line 4, and when I sees r.info ≠ r.info at line 47. Let U be the SCX-record to which this freezing CAS belongs, and S be the invocation of SCX that created U. (We prove below that S exists.) We say I blames S for r.

Lemma 70. If an invocation I of VLX returns False, then I blames some invocation of SCX.

Proof. Since I returns False, it sees r.info ≠ r.info at line 47, for some r ∈ V. Let p be the process that performs I. By line 46, r.info is a copy of r’s info value in p’s local table of LLX results. By the precondition of VLX and the definition of an LLX(r) linked to I, this value is read from r.info at line 4 by the LLX(r) linked to I. Therefore, r.info must change between when the LLX(r) linked to I performs line 4 and when I sees r.info ≠ r.info at line 47. Thus, there must be a successful freezing CAS belonging to some SCX-record U on r between these two times. Since a freezing CAS belongs to U, Lemma 11 implies that U is not the dummy SCX-record, so U must have been created by an invocation of SCX. □

Definition 71. Let U be an SCX-record created by an invocation S of SCX that returns False, and U’ be an SCX-record created by an invocation S’ of SCX. Consider the Data-records r that are in both U.V and U’.V, and for which there is no successful freezing CAS belonging to U on r. Let r’ be the Data-record among these which occurs earliest in U.V. We say S blames S’ for r’ if and only if there is a successful freezing CAS on r’ belonging to U’, and this freezing CAS is the earliest successful freezing CAS on r’ to occur between when the LLX(r’’) linked to S reads r’.info at line 4 and the first freezing CAS belonging to U on r’.

Lemma 72. Let U be an SCX-record created by an invocation S of SCX. If S returns False, then it blames some other invocation of SCX.

Proof. Since S returns False, Lemma 56.2 implies that an abort step belongs to U. By Lemma 24, there is a Data-record r_k in U.V such that there is a freezing CAS belonging to U on r_k, but no successful one. Moreover, r_k.info changes after time t_1 when the LLX(r_k) linked to S reads r_k.info at line 9, and before time t_2, when the first freezing CAS belonging to U on r_k occurs. Since the LLX(r_k) linked to S terminates before U is created (at line 21 of S), and a freezing CAS belonging to U can only occur after U is created, we know t_1 < t_2. Let t_0 be the time when the LLX(r_k) performs line 4. Note that t_0 < t_1 < t_2. Since r_k.info can only be changed by a successful freezing CAS, there must be a successful freezing CAS on r_k during (t_1, t_2). Let feas be the earliest successful freezing CAS on r_k during (t_0, t_2), and let U’ be the SCX-record to which it belongs. Since feas occurs before the first freezing CAS belonging to U on r_k, we know that U ≠ U’. Let

ρ = {r | r is in U.V and r is in U’.V and r is successful freezing CAS belonging to U on r’}

By the code of HELP, a freezing CAS belonging to U’ can only modify a Data-record in U’.V. Thus, r_k ∈ ρ.

We now show r_k is the element of ρ that occurs earliest in U.V. Suppose, to derive a contradiction, that some r_i ∈ ρ comes before r_k in U.V. By Lemma 24.1 and Lemma 24.2, there is an unsuccessful freezing CAS belonging to U on r_i. By Lemma 18, before this unsuccessful freezing CAS, there must be a successful freezing CAS belonging to U on r_i. However, this implies r_i ∈ ρ, which is a contradiction.

Let S’ be the invocation of SCX that creates U’. Thus far, we have shown that S blames S’. It remains to show that S ≠ S’. By Lemma 11, a freezing CAS or abort step cannot belong to the dummy SCX-record. Therefore, neither U nor U’ can be the dummy SCX-record. Since U ≠ U’, U and U’ must be created by different invocations of SCX. □

We now prove that an invocation of LLX(r) can return Fail only under certain circumstances.

Definition 73. Let U be an SCX-record created by an invocation S of SCX. The threatening section of S begins with the first freezing CAS belonging to U, and ends with the first commit step or abort step belonging to U.
Lemma 74. Let \( U \) be an SCX-record created by an invocation \( S \) of SCX. The threatening section of \( S \) lies within \( S \), and every successful freezing CAS or update CAS belonging to \( U \) occurs during \( S \)'s threatening section.

Proof. Let \( ptr \) be a pointer to \( U \), and \( t_0 \) and \( t_1 \) be the times when \( S \)'s threatening section begins and ends, respectively. Since each freezing CAS, update CAS, abort step or commit step belonging to \( U \) occurs in an invocation of Help\((ptr)\), and \( S \) creates \( U \), we know that these steps can only occur after \( S \) begins. Hence, \( t_0 \) is after \( S \) begins. Clearly every freezing CAS belonging to \( U \) occurs after \( t_0 \).

From the code of Help, the first update CAS belonging to \( U \) occurs between \( t_0 \) and \( t_1 \). By Lemma 47, this is the only update CAS belonging to \( U \) that can succeed. By Lemma 14, no freezing CAS belonging to \( U \) can succeed after the first frozen step or abort step belonging to \( U \). From the code of Help, the first frozen step belonging to \( U \) must occur before the first commit step belonging to \( U \). Therefore, a commit step or abort step belonging to \( U \) must occur before \( S \) terminates, so \( t_1 \) is before \( S \) terminates.

Observation 75. Let \( S \) be an invocation of SCX\((V,R,fld,new)\), and \( U \) be the SCX-record it creates. If there is a freezing CAS belonging to \( U \) on \( r \), then \( r \) is in \( V \).

Proof. From the code of Help, there will only be a freezing CAS belonging to \( U \) on \( r \) if \( r \) is in \( U \), and line 21 implies that \( r \) is in \( V \).

Lemma 76. An invocation \( I \) of LLX\((r)\) can return False only if it overlaps the threatening section of some invocation of SCX\((V,R,fld,new)\) with \( r \) in \( V \).

Proof. By Lemma 68, \( I \) blames an invocation \( S \) of SCX. Let \( U \) be the SCX-record created by \( S \), and \( ptr \) be a pointer to \( U \). By Definition 67, \( I \) reads a pointer to \( U \) from \( r.info \). Since \( U \) is not the dummy SCX-record, \( r.info \) can only point to \( U \) after a successful freezing CAS belonging to \( U \) on \( r \).

By Observation 75, \( r \) is in \( V \). We now show that \( I \) overlaps the threatening section of \( S \). Consider the two cases of Definition 67.

Case I: \( I \) enters the if-block at line 7, and reads a pointer to \( U \) from \( r.info \) at line 9. In this case, from the code of LLX, we know that \( r.info \) changes between when \( I \) performs line 4 and when \( I \) performs line 9. Since \( r.info \) can only be changed to point to \( U \) by a successful freezing CAS belonging to \( U \), there must be a successful freezing CAS belonging to \( U \) during \( I \). By Lemma 74, \( I \) must overlap the threatening section of \( S \).

Case II: \( I \) does not enter the if-block at line 7. Since \( I \) reads a pointer to \( U \) from \( r.info \) at line 4, we know that \( r.info \) is a pointer to \( U \). By the test at line 7, either \( state = Committed \) and \( marked = True \), or \( state =InProgress \).

Suppose \( state = InProgress \). Then, \( U.state = InProgress \) when \( I \) performs line 5. Since \( U \) is not the dummy SCX-record, a pointer to \( U \) can appear in \( r.info \) only after a successful freezing CAS belonging to \( U \). By Corollary 23, \( U.state \) can only be InProgress before the first commit step or abort step belonging to \( U \). Therefore, Definition 73 implies that \( I \) performs line 5 during the threatening section of \( S \).

Now, suppose \( state = Committed \) and \( marked = True \). By Corollary 23, \( U.state = Committed \) at all times after \( I \) performs line 5. We consider two sub-cases. If \( marked = True \), then \( I \) will return Finalized if it reaches line 12. Since we have assumed that \( I \) returns Fail, this case is impossible. Otherwise, a mark step mstep belonging to some SCX-record \( W \) changes \( r.marked \) to True between line 3 and line 6. It remains only to show that mstep occurs during the threatening section of the invocation of SCX that created \( W \). Since \( r.marked \) is initially False, and is never changed from True to False, mstep must be the first mark step belonging to \( W \) on \( r \). From the code of Help, a frozen step belonging to \( W \) must precede mstep. Therefore, Lemma 21 implies that no abort step belonging to \( W \) ever occurs. From the code of Help, mstep must occur after the first freezing CAS belonging to \( W \), and before the first commit step belonging to \( W \). By Definition 73, mstep occurs during the threatening section of the invocation of SCX that created \( W \).

We now prove that an invocation of SCX or VLX can return False only under certain circumstances.

Definition 77. The vulnerable interval of an invocation \( I \) of SCX or VLX begins at the earliest starting time of an LLX\((r)\) linked to \( I \), and ends when \( I \) ends.

Lemma 78. Let \( I \) be an invocation of SCX or VLX, and \( U \) be an SCX-record created by an invocation \( S \) of SCX. If \( I \) blames \( S \) for a Data-record \( r \), then a successful freezing CAS belonging to \( U \) on \( r \) occurs during \( I \)'s vulnerable interval.

Proof. Suppose \( I \) is an invocation of SCX. Let \( U_1 \) be the SCX-record created by \( I \). By Definition 71, a successful freezing CAS belonging to \( U_1 \) on \( r \) occurs between when the LLX\((r)\) linked to \( I \) performs line 4, and the first freezing CAS fcas belonging to \( U_1 \) on \( r \). By Lemma 74, fcas occurs during \( I \). Now, suppose \( I \) is an invocation of VLX. Then, by Definition 69, a successful freezing CAS belonging to \( U_1 \) on \( r \) occurs between when the LLX\((r)\) linked to \( I \) performs line 4, and when \( I \) sees \( r.info \neq r.info \) at line 9.

Observation 79. If an invocation \( I \) of SCX\((V,R,fld,new)\) or VLX\((V)\) blames an invocation of SCX for a Data-record \( r \), then \( r \) is in \( V \).

Proof. Suppose \( I \) is an invocation of SCX. Let \( U \) be the SCX-record created by \( I \). By Definition 71, \( r \) is in \( U \). Since \( U \) does not change after \( U \) is created at line 21 of \( I \), \( r \) is in \( V \). Now, suppose \( I \) is an invocation of VLX. In this case, the claim is immediate from Definition 69.

Observation 80. If an invocation \( I \) of SCX or VLX blames an invocation \( S \) of SCX\((V,R,fld,new)\) for a Data-record \( r \), then \( r \) is in \( V \).

Proof. Let \( U \) be the SCX-record created by \( S \). By Lemma 78, there is a successful freezing CAS belonging to \( U \) on \( r \). The claim then follows from Observation 75.

Lemma 81. An invocation \( I \) of SCX\((V,R,fld,new)\) or VLX\((V)\) ending at time \( t \) can return False only if its vulnerable interval overlaps the threatening section of some other SCX\((V',R',fld',new')\), where some Data-record appears in both \( V \) and \( V' \).
Proof. Suppose \( I \) returns False. By Lemma 70 and Lemma 72, \( I \) blames an invocation \( S \) of SCX(\( V', R', \text{fld}', \text{new}' \)), where \( I \neq S \), for a Data-record \( r \). Let \( U \) be the SCX-record created by \( S \), and \( U_I \) be the SCX-record created by \( I \). By Lemma 78, a successful freezing CAS \( \text{cas} \) belonging to \( U \) on \( r \) occurs during \( I \)'s vulnerable interval. By Lemma 74, \( \text{cas} \) occurs during the threatening section of \( S \). Therefore, \( I \)'s vulnerable interval overlaps the threatening section of \( S \). By Observation 79 and Observation 80, \( r \) is in both \( V \) and \( V' \). □

We now prove bounds on the number of invocations of LLX, SCX and VLX that can blame an invocation of SCX.

Lemma 82. Let \( I \) be an invocation of LLX(\( r \)) that returns \text{Fail}, and \( U \) be the SCX-record created by the invocation of SCX that is blamed by \( I \). \( A \) commit step or abort step belonging to \( U \) occurs before \( I \) returns.

Proof. By Definition 67, we know that \( I \) reads a pointer to \( U \) from \( r.info \). By Lemma 68, \( U \) is not the dummy SCX-record. This implies that a pointer to \( U \) can only appear in \( r.info \) after a successful freezing CAS belonging to \( U \) on \( r \). By Corollary 26, \( r.info \) points to \( U \) at all times after the first freezing CAS belonging to \( U \) on \( r \), and before the first commit step or abort step belonging to \( U \). Thus, if \( r.info \) changes after a pointer to \( U \) is read from \( r.info \) at line 4 or line 9, and before either of the two times that \( r.info \) is read at line 15, then we know that a commit step or abort step belonging to \( U \) has already occurred. Otherwise, any read of \( r.info \) at line 15 returns a pointer to \( U \). Hence, \( I \) checks whether \( U.state = \text{InProgress} \) at line 15 and, if so, \( I \) helps \( U \). We consider two cases.

Case I: \( I \) sees \( U.state = \text{InProgress} \) at line 15. In this case, \( I \) helps \( U \) before returning. From the code of \( \text{HELP} \), if \( I \)'s invocation of \( \text{HELP} \) returns \text{False}, then \( I \) performs an abort step belonging to \( U \) during its invocation of \( \text{HELP} \). Otherwise, by Lemma 55.3, a commit step belonging to \( U \) occurs before \( I \)'s invocation of \( \text{HELP} \) returns.

Case II: \( I \) sees \( U.state \neq \text{InProgress} \) at line 15. In this case, \( U.state \) must be Committed or Aborted. Since \( U \) is not the dummy SCX-record, we know that \( U.state \) is initially \text{InProgress}. Therefore, a commit step or abort step belonging to \( U \) must occur before line 15. □

Lemma 83. Each invocation of SCX can be blamed by at most two invocations of LLX per process.

Proof. Let \( S \) be an invocation of SCX. To derive a contradiction, suppose there is some process \( p \) that blames \( S \) for three failed invocations of LLX, \( I''_1, I''_2 \) and \( I \) (which are performed by \( p \) in this order). By Definition 67, \( I, I' \) and \( I'' \) each read a pointer to \( U \) from \( r.info \), either at line 4 or at line 9. Since \( r.info \) points to \( U \) at some point during \( I''_1 \), and again at or after the time \( I''_1 \) performs line 4, we know from Lemma 12 that \( r.info \) points to \( U \) when \( I''_1 \) performs line 4. By the same argument, \( r.info \) points to \( U \) when \( I''_2 \) performs line 4. Thus, in both \( I''_1 \) and \( I''_2 \), the local variable \( r.info \) is a pointer to \( U \).

By Lemma 82, a commit step or abort step belonging to \( U \) occurs prior to the termination of \( I'' \), which is before the start of \( I' \). By Corollary 23, \( r.info.state \) does not change after it is set to Committed or Aborted by this commit step or abort step. Since \( I''_1 \) returns \text{Fail}, when \( I''_1 \) performs line 12, either \( r.info.state = \text{Commited} \) and \( \text{marked}_1 = \text{False} \), or \( r.info.state = \text{Aborted} \).

Suppose \( r.info.state = \text{Aborted} \) when \( I' \) performs line 12. Then, \( r.info.state \) was Aborted when \( I' \) performed line 5, and \( I' \) passed the test at line 7, and entered the if-block. Since we have assumed that \( I''_1 \) blames \( S \), \( I''_1 \) must read a pointer to \( U \) from \( r.info \) when it performs line 9. However, since \( I''_1 \) returns \text{Fail}, the value \( r.info \) reads at line 9 is different from the value read at line 4, which is a contradiction. Hence, this case is impossible.

Now, suppose \( r.info.state = \text{Commited} \) and \( \text{marked}_1 = \text{False} \) when \( I' \) performs line 12. Then, \( r.info.state \) was Commited at all times since \( I''_1 \) began, so \( state = \text{Commited} \). If \( \text{marked}_1 \) was also \text{False} when \( I''_1 \) performed line 7, then \( I''_1 \) passed the test at line 7, and entered the if-block, so we obtain the same contradiction as above. Hence, this case, too, is impossible. Thus, \( \text{marked}_1 \) must have been True when \( I''_1 \) performed line 7, which means \( r \) was marked when \( I''_1 \) performed line 6. Since a commit step belonging to \( U \) had already occurred when \( I''_1 \) performed line 3, the code of \( \text{HELP} \) implies that, if \( r \) were in \( U.R \), then the first mark step belonging to \( U \) on \( r \) would already have occurred before \( I''_1 \) performed line 3. Since a marked bit is never changed from \text{True} to \text{False}, \( r \) would be marked when \( I''_1 \) performed line 3, which is incompatible with our assumption that \( \text{marked}_1 = \text{False} \). Therefore, \( r \) is not in \( U.R \), so no mark step belonging to \( U \) on \( r \) can ever occur. Since \( r \) was marked when \( I''_1 \) performed line 6, there must have been a successful mark step \( \text{mstep} \) belonging to some other SCX-record \( W \) on \( r \) after \( I''_1 \) performed line 3, and before \( I''_1 \) performed line 6. By Lemma 34, \( r.info \) points to \( W \) when \( \text{mstep} \) occurs. However, since \( r.info \) points to \( U \) during \( I''_1 \), which is before \( \text{mstep} \), and again during \( I \), which is after \( \text{mstep} \), at some point after \( \text{mstep} \), \( r.info \) must be changed to a value \( U \) that has previously appeared there, which contradicts Lemma 12. Thus, it is impossible for three or more invocations of LLX by \( p \) to blame the same invocation of SCX. □

Lemma 84. Each invocation of SCX(\( V,R,\text{fld},\text{new} \)) can be blamed by at most \(|V|\) invocations of SCX or VLX per process.

Proof. By Observation 80, if an invocation of SCX or VLX blames an invocation of SCX(\( V,R,\text{fld},\text{new} \)) for \( r \), then \( r \) is in \( V \). Thus, it suffices to prove that an invocation of SCX(\( V,R,\text{fld},\text{new} \)) cannot be blamed for any \( r \) in \( V \) by more than one invocation of SCX or VLX performed by process \( p \).

Let \( I \) and \( I' \) be invocations of SCX or VLX performed by process \( p \), and \( U, U_1 \) and \( U_2 \) be the SCX-records created by \( I \), \( I' \) and \( S \), respectively. Without loss of generality, let \( I' \) occur after \( I \). Suppose, in order to derive a contradiction, that \( I \) and \( I' \) both blame \( S \) for the same Data-record \( r \). Let \( t_0 (t_0') \) be the time when the LLX(\( r \)) linked to \( I' \) (\( I' \)) performs line 4, and \( t_1 (t_1') \) be the time when \( I' \) (\( I' \)) finishes. By Lemma 78, a successful freezing CAS belonging to \( U_2 \) occurs between \( t_0 \) and \( t_1 \), and a successful freezing CAS belonging to \( U_2 \) occurs between \( t_0' \) and \( t_1' \). If we can show \( t_0 < t_1 < t_0' < t_1' \), then we shall have demonstrated that there must be two such freezing CASs, which contradicts Lemma 14.

Since the LLX(\( r \)) linked to \( I' \) terminates before \( I' \) finishes, we know \( t_0 < t_1 \). By Observation 79, \( r \) is in \( V \) when \( r \) is in the \( V \) sequences of invocations \( I \) and \( I' \). Hence, Definition 7.2 implies that \( t_1 \notin [t_0,t_1'] \) and \( t_0' \notin [t_0,t_1] \). Since \( I' \) occurs
after $I$, $t_1 < t'_1$. Therefore, $t_0 < t_1 < t'_0 < t'_1$. □

We now define the blame graph and prove a number of its properties.

**Definition 85.** We define the blame graph for an execution to be a directed graph whose nodes are the invocations of LLX, VLX and SCX, with an edge from an invocation $I$ to another invocation $I'$ if and only if $I$ blames $I'$. (Note that only the nodes corresponding to invocations of SCX can have incoming edges.)

The next property we prove is that, for each execution, there is a bound on the length of the longest path in the blame graph. As mentioned in Section 3, we require the following constraint in order to prove this bound exists.

**Constraint 86.** If there is a configuration $C$ after which the value of no field of any Data-record changes, then there is a total order $\prec$ on all Data-records created during the execution such that, if Data-record $r_1$ appears before data Data-record $r_2$ in the sequence $V$ passed to an invocation of SCX whose linked LLXs begin after $C$, then $r_1 \prec r_2$.

**Lemma 87.** Let $U$ be an SCX-record created by an invocation of SCX whose linked LLXs begin after the configuration $C$ that is specified in Constraint 86. Immediately after a successful freezing CAS belonging to $U$ on $r$, $r$.info points to $U$ and, for each $r'$ in $U.V$, where $r' \prec r$, $r'$.info points to $U$ and a successful freezing CAS belonging to $U$ on $r'$ has occurred.

**Proof.** Let $fcas$ be a successful freezing CAS belonging to $U$ on $r$, and let $r'$ be any Data-record in $U.V$ that satisfies $r' \prec r$. By Constraint 86, $r'$ must occur before $r$ in the sequence $U.V$. By Lemma 18, a successful freezing CAS $fcas'$ belonging to $U$ on $r'$ occurs prior to $fcas$. Thus, Corollary 26 implies that $r'$.info points to $U$ at all times after $fcas'$ and before the first commit step or abort step belonging to $U$. Similarly, $r$.info points to $U$ at all times after $fcas$ and before the first commit step or abort step belonging to $U$. By Lemma 25, $fcas$ must precede the first frozen step or abort step belonging to $U$. From the code, an update CAS can only occur in an invocation of HELP($ptr$), where $ptr$ points to an SCX-record $U$. Further, from the code of HELP, there is at least one freezing CAS belonging to $U$ or frozen step belonging to $U$. Hence, Lemma 11 implies that $U$ is not the dummy SCX-record. Thus, $U$ is created by an invocation of SCX at line 21. By Lemma 47, only the first update CAS belonging to an SCX-record can succeed. By Lemma 56.3, no update CAS belongs to an SCX-record created by an unsuccessful invocation of SCX. □

We think of processes as accessing such a data structure via a fixed number of special Data-records called entry points, each of which has a single mutable pointer to a Data-record. We assume there is always some Data-record reachable by following pointers from an entry point that is not finalized. (This assumption that entry points cannot be finalized is not crucial, but it simplifies the statement of some progress guarantees.)

**Definition 90.** A Data-record is initiated at all times after it first becomes reachable by following Data-record pointers from an entry point.

**Observation 91.** The only step in an execution that can cause a Data-record to become initiated is a successful update CAS.

**Proof.** Follows immediately from Observation 38. □

**Lemma 92.** Let $S_1$ and $S_2$ be invocations of SCX, and let $r$ be a Data-record. If $S_1$ blames $S_2$ for $r$, then $r$ was initiated before the start of $S_1$, and before the start of $S_2$.

**Proof.** Let $U_1$ and $U_2$ be the SCX-records created by $S_1$ and $S_2$, respectively. By Definition 71, $r$ is in both $U_1.V$ and $U_2.V$. By Observation 8.3, there are invocations of LLX($r$) linked to $S_1$ and $S_2$, respectively. By the precondition of LLX, $r$ must be initiated before the LLX($r$) linked to $S_1$, and before the LLX($r$) linked to $S_2$. Finally, the LLX($r$) linked to $S_1$ must terminate before $S_1$ begins, and the LLX($r$) linked to $S_2$ must terminate before $S_2$ begins. □

**Lemma 93.** If no SCX is linearized after some time $t$, then the following hold.

1. A finite number $N$ of Data-records are ever initiated in the execution.

2. Let $\sigma$ be the set of invocations of SCX in the execution whose vulnerable intervals start at or before $t$. The longest path in the blame graph consisting entirely of invocations of LLX, SCX, and VLX that are not in $\sigma$ has length at most $N+2$

**Proof.** Claim 1 follows immediately from Observation 91 and Lemma 89.

We now prove claim 2. Suppose, in order to derive a contradiction, that there is a path of length at least $N+3$ in the blame graph consisting entirely of invocations of LLX, SCX, and VLX that are not in $\sigma$. Since only invocations of SCX can be blamed, at least $N+2$ of the nodes on this path must correspond to invocations of SCX. Let $S_1, S_2, \ldots, S_{N+2}$ be invocations of SCX corresponding to any $N+2$ consecutive nodes on this path, and let $U_1, U_2, \ldots, U_{N+2}$ be the SCX-records they created, respectively. For each
Let \( i \in \{1, 2, \ldots, N + 1\} \), let \( r_i \) be the Data-record for which \( S_i \) blames \( S_{i+1} \). Since no invocation of SCX is linearized after \( t \), and the vulnerable sections of \( S_1, S_2, \ldots, S_{N+2} \) all start after \( t \), no invocation of SCX is linearized after the first LLX(\( r \)) linked to any of these invocations of SCX. Therefore, from Lemma 88 and the fact that, for each \( i \in \{1, 2, \ldots, N\} \), \( S_i \) blames \( S_{i+1} \) for \( r_i \), and \( S_{i+1} \) blames \( S_{i+2} \) for \( r_{i+1} \), we obtain \( r_i \prec r_{i+1} \). By Lemma 92, before any invocation of SCX in \( \{S_1, S_2, \ldots, S_{N+2}\} \), \( r_1, r_2, \ldots, r_{N+1} \) have all been initiated. Therefore, some Data-record \( r \) appears twice in \( \{r_1, r_2, \ldots, r_{N+1}\} \). Since the \( \prec \) relation is transitive, we obtain \( r \prec r \), which is a contradiction. 

We now prove the main progress property for SCX.

**Lemma 94.** If invocations of SCX complete infinitely often, then invocations of SCX succeed infinitely often.

**Proof.** Suppose, to derive a contradiction, that after some time \( t' \), invocations of SCX are performed infinitely often, but no invocation of SCX is successful. Then, since we only linearize successful SCXs, and a subset of the non-terminating SCXs, there is a time \( t \geq t' \) after which no invocation of SCX is linearized. Let \( \sigma \) be the set of invocations of SCX in the execution whose vulnerable intervals start at or before \( t \). By Lemma 83 and Lemma 84, the in-degree of each node in the blame graph is bounded. Since \( \sigma \) is finite, and the in-degree of each node in \( \sigma \) is bounded, only a finite number of invocations of SCX can blame invocations in \( \sigma \). Now, consider any maximal path \( \pi \) consisting entirely of invocations of LLX, SCX, and VLX that are not in \( \sigma \). By Lemma 93.2, \( \pi \) has length at most \( N + 3 \). Since no invocation of SCX is successful after \( t \), the invocation \( S \) of SCX corresponding to the last node on path \( \pi \) must be unsuccessful. By Lemma 72, \( S \) must blame some other invocation of SCX. Since \( \pi \) is maximal, \( S \) must blame an invocation of SCX in \( \sigma \). Thus, there can be only finitely many of these paths (of bounded length). However, this contradicts our assumption that invocations of SCX occur infinitely often. 

Unfortunately, since SCX cannot be invoked unless a sequence of invocations of LLX (linked to this SCX) return values different from Fail or Finalized, the previous result is not strong enough unless we can guarantee that processes can invoke SCX infinitely often. To address this, we define setting up an invocation of SCX.

**Definition 95.** A process \( p \) **sets up** an invocation of SCX(V, R, fld, new) by invoking LLX(\( r \)) for each \( r \) in \( V \), and then invoking SCX(V, R, fld, new) if none of these LLXs return Fail or Finalized.

**Theorem 96.** Our implementation of LLX/SCX/VLX satisfies the following progress properties.

1. If operations (LLX, SCX, VLX) are performed infinitely often, then operations succeed infinitely often.

2. If invocations of SCX are set up infinitely often, then invocations of SCX succeed infinitely often.

3. If there is always some Data-record reachable by following pointers from an entry point that is not finalized, then invocations of SCX can be set up infinitely often.

**Proof.** Claim 3 is obvious. The first two claims have similar proofs, by cases.

**Proof of Claim 1.** Suppose operations are performed infinitely often.

**Case I:** invocations of SCX are performed infinitely often. In this case, Lemma 94 implies that invocations of SCX will succeed infinitely often, and the claim is proved.

**Case II:** after some time \( t \), no invocation of SCX is performed. In this case, the blame graph contains a finite number of invocations of SCX. By Lemma 83 and Lemma 84, the in-degree of each node in the blame graph is bounded. By Lemma 68 and Lemma 70, each unsuccessful invocation of LLX or VLX blames an invocation of SCX. Therefore, only finitely many invocations of LLX and VLX can be unsuccessful. Thus, eventually, every invocation of LLX or VLX succeeds.

**Proof of Claim 2.** Suppose invocations of SCX are set up infinitely often.

**Case I:** invocations of SCX are performed infinitely often. In this case, Lemma 94 implies that invocations of SCX will succeed infinitely often, and the claim is proved.

**Case II:** after some time \( t \), no invocation of SCX is performed. Suppose, to derive a contradiction, that after some time \( t \), SCX is never invoked. Then, as we argued in Case II, above, only finitely many invocations of LLX and VLX can be unsuccessful. This implies that, after some time \( t' \), every invocation of LLX is successful. If a process \( p \) begins setting up an invocation of SCX after \( t' \), then all of its invocations of LLX will be successful, and the only way that \( p \) will invoke SCX is if some invocation of LLX(\( r \)) by \( p \) returns Finalized. By Definition 95, \( p \) will not invoke LLX(\( r \)) next time it sets up an invocation of SCX. By Lemma 93, there are a finite number \( N \) of Data-records that are ever initiated in the execution. Therefore, eventually, \( p \) will have performed an invocation of LLX(\( r' \)) that returned Finalized for every Data-record \( r' \) that is ever initiated in the execution. After this, \( p \) will no longer be able to set up invocations of SCX. This contradicts our assumption that invocations of SCX are set up infinitely often. 


B. ADDITIONAL PROPERTIES OF LLX/SCX/VLX

In this section we prove some additional properties of LLX/SCX/VLX that are intended to simplify the design of certain data structures. At this level, a configuration consists of the state of each process, and a collection of Data-records (which have only mutable and immutable fields). A step is either a READ, or a linearized invocation of LLX, SCX or VLX.

**Definition 97.** A Data-record r is in the data structure in some configuration C if and only if r is reachable by following pointers from an entry point. We say a Data-record r is removed (from the data structure) by some step s if and only if r is in the data structure immediately before s, and r is not in the data structure immediately after s. We say a Data-record r is added (to the data structure) by some step s if and only if r is not in the data structure immediately before s, and r is in the data structure immediately after s.

Note that a Data-record can be removed from or added to the data structure only by a linearized invocation of SCX.

If the following constraint is satisfied, then the results of this section apply.

**Constraint 98.** For each linearized invocation S of SCX(V, R, fld, new), R contains precisely the Data-records that are removed from the data structure by S.

**Lemma 99.** If a Data-record r is removed from the data structure for the first time by step s, then no linearized invocation of SCX(V, R, fld, new), where fld is a mutable field of r, occurs at or after s. (Hence, r does not change at or after s.)

**Proof.** The only step that can change r is a linearized invocation of SCX. The invocation S′ of SCX(V′, R′, fld, new′) that removes r modifies a mutable field of some Data-record different from r. Thus, fld′ is not a field of r. Since this is the only change to the data structure when s occurs, r does not change when s occurs. Suppose, to derive a contradiction, that an invocation S of SCX(V, R, fld, new), where fld is a mutable field of r, occurs after s. Then, since r is in V, the precondition of SCX implies that an invocation I of LLX(r) linked to S must occur before S. By Constraint 98, r is in R′. Thus, if I occurs after S, then it returns FINALIZED, which contradicts Definition 7. Otherwise, S′ occurs between I and S, so S cannot be linearized, which contradicts our assumption.

**Lemma 100.** If an invocation I of LLX(r) returns a value different from FAIL or FINALIZED, then r is in the data structure just before I is linearized.

**Proof.** By the precondition of LLX, r is initiated and, hence, in the data structure, at some point before I. Suppose, to derive a contradiction, that r is not in the data structure just before I is linearized. Then, some linearized invocation of SCX(V, R, fld, new) must remove r before I is linearized. By Constraint 98, r is in R. However, this implies that I must return FINALIZED, which is a contradiction.

**Lemma 101.** If S is a linearized invocation of SCX(V, R, fld, new), where new is a Data-record, then new is in the data structure just after S.

**Proof.** Note that fld is a mutable field of a Data-record r in V. We first show that r is in the data structure at some point before S. By the precondition of SCX, before S, there is an LLX(r) linked to S. By the precondition of LLX, r must be initiated when this linked LLX occurs. Thus, Definition 90 and Definition 97 imply that r is in the data structure at some point before S. Suppose, to derive a contradiction, that r is not in the data structure just after S. Then, r must either be removed by S, or by some previous step. However, this directly contradicts Lemma 99.

Let C1 and C2 be configurations in the execution. We use C1 < C2 to mean that C1 precedes C2 in the execution. We say C1 ≤ C2 precisely when C1 = C2 or C1 < C2. We denote by [C1, C2] the set of configurations {C | C1 ≤ C ≤ C2}.

**Lemma 102.** Let r1, r2, ..., rl be a sequence of Data-records, where r1 is an entry point, and C1, C2, ..., Cl−1 be a sequence of configurations satisfying C1 < C2 < ... < Cl−1. If, for each i ∈ {1, 2, ..., l − 1}, a field of ri points to r i+1 in configuration Ci, then r i+1 is in the data structure in some configuration in [Ci, Ci+1]. Additionally, if a mutable field f of ri contains a value v in some configuration C after Ci then, in some configuration in [Ci, Ci+1], r1 is in the data structure and f contains v.

**Proof.** We prove the first part by induction on i.

Since each entry point is always in the data structure, and r1 points to r2 in configuration C1, r2 is in the data structure in C1. Thus, the claim holds for i = 1.

Suppose the claim holds for i, 1 ≤ i ≤ l − 1. We prove it holds for i + 1. If ri is in the data structure when it points to ri+1 in Ci, then ri+1 is in the data structure in Ci, and we are done. Suppose ri is not in the data structure in Ci. By the inductive hypothesis, ri is in the data structure in some configuration in [Ci, Ci+1]. Let t, C1 < t < Ci, be the first step such that ri is removed from the data structure by S. In the configuration C just before s, ri is in the data structure. By Lemma 99, ri does not change at or after s. Thus, ri does not change after C. Since C occurs before Ci, and ri points to ri+1 in Ci, ri must point to ri+1 in C. Therefore, in (which satisfies C1 ≤ C < C2), ri is in the data structure and points to ri+1.

The second part of the proof is quite similar to the inductive step we just finished. Suppose f contains v in Ci. If ri is in the data structure in Ci, then we are done. Suppose r i is not in the data structure in Ci. We have shown above that ri is in the data structure in some configuration in [Ci, Ci+1]. Let s′, C1 < s′ < Ci, be the first step such that ri is removed from the data structure by s′. In the configuration C′ just before s′, ri is in the data structure. By Lemma 99, ri does not change at or after s′. Thus, ri does not change after C′. Since C′ occurs before Ci and f contains v in Ci, f must contain v in C′. Therefore, in C′ (which satisfies C1 ≤ C′ < C2), ri is in the data structure and f contains v.
C. COMPLETE PROOF OF MULTISET IMPLEMENTATION

The full pseudocode for the multiset algorithm appears in Fig. 6. Initially, the data structure contains a head entry point, containing a single mutable field next that points to a sentinel Node with a special key $\infty$ that is larger than any key that can appear in the multiset.

In the following, we define the response of a Search to be a step at which a value is returned. Note that we specify Lemma 103.3, instead of directly proving the considerably simpler statement in Constraint 98, so that we can reuse the intermediate results when proving linearizability.

**Lemma 103.** The multiset algorithm satisfies the following properties.

1. Every invocation of LLX or SCX has valid arguments, and satisfies its preconditions.
2. Every invocation of SEARCH satisfies its postconditions.

3. Let $S$ be an invocation of SCX($V, R, fld, new$) performed by an invocation $I$ of INSERT or DELETE, and $p, r$ and $r.next$ refer to the local variables of $I$. If $I$ performs $S$ at line 20, then no Data-record is added or removed by $S$, and $R = \emptyset$. If $I$ performs $S$ at line 24, then only new is added by $S$, no Data-record is removed by $S$, and $R = \emptyset$. If $I$ performs $S$ at line 34, then only new is added by $S$, only $r$ is removed by $S$, and $R = \{r\}$. If $I$ performs $S$ at line 37, then only new is added by $S$, only $r$ and $r.next$ are removed by $S$, and $R = \{r, r.next\}$.

4. The head entry point always points to a Node, the next pointer of each Node with key $\neq \infty$ points to some Node with a strictly larger key, and the next pointer of each Node with key $\infty$ is Nil.

**Proof.** We prove these claims by induction on the sequence of steps taken in the execution. The only steps that can affect these claims are invocations of LLX and SCX, and responses of SEARCHES. **Base case.** Clearly, Claim 1, Claim 2 and Claim 3 hold before any such step occurs. Before the first SCX, the data structure is in its initial configuration. Thus, Claim 4 holds before any step occurs. **Inductive step.** Suppose these claims hold before some step $s$. We prove they hold after $s$.

**Proof of Claim 1.** The only steps that can affect this claim are invocations of LLX and SCX.

Suppose $s$ is an invocation of LLX. By inductive Claim 3 and Observation 105, Constraint 98 is satisfied at all times before $s$ occurs. The only places in the code where $s$ can occur are at lines 18, 22, 29, 30 and 36. Suppose $s$ occurs at line 18, 22, 29 or 30. Then, by inductive Claim 2, argument to $s$ is non-Nil. We can apply Lemma 102 to show that the argument to $s$ is in the data structure and, hence, initiated, at some point during the last SEARCH before $s$. Now, suppose $s$ occurs at line 36 (so localr.next is the argument to $s$). Then, key = $r.key$ when line 32 is performed so, by the precondition of DELETE, $r.key \neq \infty$. By inductive Claim 4, $r.next \neq \text{Nil}$ when LLX($r$) is performed at line 30, so localr.next $\neq \text{Nil}$. We can apply Lemma 102 to show that localr.next is in the data structure and, hence, initiated, at some point between the start of the last SEARCH before $s$ and the last LLX($r$) before $s$ (which reads localr.next from $r.next$).

Suppose $s$ is a step that performs an invocation $S$ of SCX($V, R, fld, new$). Then, the only places in the code where $s$ can occur are at lines 20, 24, 34 and 37. It is a trivial exercise to inspect the code of INSERT and DELETE, and argue that the process that performs $s$ has done an LLX($r$) linked to $S$ for each $r \in V$, that $R \subseteq V$, and that $fld$ points to a mutable field of a Data-record in $V$. It remains to prove that Precondition (2) and Precondition (3) of SCX are satisfied. Let $I$ be the invocation of LLX($r$) linked to $S$. Suppose $s$ occurs at line 20. The only step that can affect the claim is a linearized invocation $s$ of SCX($V, R, fld, new$) performed at line 20. From the code, $fld$ is $r.count$. Since $s$ is linearized, no invocation of SCX($V', R', fld', new'$) with $r \in V'$ is linearized between $I$ and $s$. Thus, $r.count$ does not change between when $I$ and $s$ are linearized. From the code, $new$ is $count$ plus the value read from $r.count$ by $I$. Therefore, $new$ is strictly larger than $r.count$ was when $I$ was linearized, which implies that $new$ is strictly larger than $r.count$ when $s$ is linearized. This immediately implies Precondition (2) and Precondition (3) of SCX. Now, suppose $s$ occurs at line 24, 34 or 37. Then, $new$ is a pointer to a Node that was created after $I$. Thus, no invocation of SCX($V', R', fld, new$) can even begin before $I$. We now prove that $new$ is not the initial value of the field pointed to by $fld$. From the code, $fld$ is $p.next$. If $p.next$ is initially Nil, then we are done. Otherwise, $p.next$ initially points to some Node $r'$. Clearly, $r'$ must be created before $p$. Hence, $r'$ must be created before the invocation of SEARCH followed a pointer to $p$. Since $new$ is a pointer to a Node that is created after this invocation of SEARCH, $new \neq r'$.

**Proof of Claim 2.** To affect this claim, $s$ must be the response of an invocation of SEARCH($key$). We prove a loop invariant that states $r$ is a Node, and either $p$ is a Node and $p.key < key$ or $p = \text{head}$. Before the loop, $p = \text{head}$ and $r = \text{head}.next$. By inductive Claim 4 head.next is always a Node, so the claim holds before the loop. Suppose the claim holds at the beginning of an iteration. Let $r$ and $p$ be the respective values of local variables $r$ and $p$ at the beginning of the iteration, and $r'$ and $p'$ be their values at the end of the iteration. From the code, $p' = r'$ and $r'$ is the value read from $r.next$ at line 12. By the inductive hypothesis, $p'$ is a Node. Since the loop did not exit before this iteration, $key > p'.key$. Further, since SEARCH($key$) is invoked only when $key < \infty$ (by inspection of the code and preconditions), $p'.key < \infty$. By inductive Claim 4, $p'.next = r.next$ always points to a Node, so $r'$ is a Node, and the inductive claim holds at the end of the iteration. Finally, the exit condition of the loop implies $key \leq r'.key$, so SEARCH satisfies its postcondition.

**Proof of Claim 3.** Since a Data-record can be removed from the data structure only by a change to a mutable field of some other Data-record, this claim can be affected only by linearized invocations of SCX. Suppose $s$ is a linearized invocation of SCX($V, R, fld, new$). Then, $s$ can occur only at line 20, 24, 34 or 37. Let $I$ be the invocation of INSERT or DELETE in which $s$ occurs. We proceed by cases.

Suppose $s$ occurs at line 20. Then, $fld$ is a pointer to $r.count$. Thus, $s$ changes a count field, not a next pointer. Since this is the only change that is made by $s$, no Data-record is removed by $s$, and no Data-record is added by $s$. Since $R = \emptyset$, the claim holds.

Suppose $s$ occurs at line 24. Then, $fld$ is a pointer to $p.next$, and $new$ is a pointer to a new Node. Before per-
forming $s$, $I$ performs an invocation $L_1$ of LLX($p$), which returns a value different from FAIL, or FINALIZED, at line 29. Just after performing $L_1$, $I$ sees that localp.$next = r$. Note that $L_1$ is linked to $s$. Since $s$ is linearized, and $p \in V$, $p.next$ does not change in between when $L_1$ and $s$ are linearized. Therefore, $s$ changes $p.next$ from $r$ to point to a new Node whose $next$ pointer points to $r$. Since $s$ is linearized, Lemma 99 implies that $p$ must be in the data structure just before $s$ (and when its change occurs). Since this is the only change that is made by $s$, no Data-record is removed by $s$, and new pointers to the only Data-record that is added by $s$. Since $R = \emptyset$, the claim holds.

Suppose $s$ occurs at line 34 or line 37. Then, $fld$ is a pointer to $p.next$, and new is a pointer to a new Node. Before performing $s$, $I$ performs invocations $L_1$ and $L_2$ of LLX($p$) and LLX($r$), respectively, which each return a value different from FAIL, or FINALIZED. Note that $L_1$ and $L_2$ are linked to $s$. Just after performing $L_1$, $I$ sees that localp.$next = r$. Since $s$ is linearized, and $p \in V$, $p.next$ does not change in between when $L_1$ and $s$ are linearized. Similarly, since $r \in V$, $r.next$ does not change between when $L_1$ and $s$ are linearized. Before $s$, $I$ sees $key = r.key$ at line 32. By the precondition of DELETE, $r.key \neq \infty$. Thus, inductive Claim 4 (and the fact that keys do not change) implies that $r.next$ points to some Node $rnext = localr.next$ at all times between when $L_2$ and $s$ are linearized. We consider two sub-cases.

Case I: $s$ occurs at line 34. Therefore, $s$ changes $p.next$ from $r$ to point to a new Node whose $next$ pointer points to $rnext$ and, when this change occurs, $r.next$ points to $rnext$. Since $s$ is linearized, Lemma 99 implies that $p$ must be in the data structure just before $s$ (and when its change occurs). Since this is the only change that is made by $s$, $r$ points to the only Data-record that is removed by $s$, and new pointers to the only Data-record that is added by $s$. Since $R = \{r\}$, the claim holds.

Case II: $s$ occurs at line 37. Since $rnext \in V$, $rnext.next$ does not change between when the LLX at line 36 and $s$ are linearized. Thus, $rnext.next$ contains the same value $v$ throughout this time. Therefore, $s$ changes $p.next$ from $r$ to point to a new Node whose $next$ pointer contains $v$ and, when this change occurs, $p.next$ points to $r$, $r.next$ points to $rnext$, and $rnext.next$ contains $v$. Since $s$ is linearized, Lemma 99 implies that $p$ must be in the data structure just before $s$ (and when its change occurs). Since this is the only change that is made by $s$, $r$ and $rnext$ point to the only Data-records that are removed by $s$, and new pointers to the only Data-record that is added by $s$. Since $R = \{r, rnext\}$, the claim holds.

Proof of Claim 4. This claim can be affected only by a linearized invocation of SCX that changes a $next$ pointer. Suppose $s$ is a linearized invocation of SCX($V, R, fld, new$). Then, $s$ can occur only at line 24, 34 or 37. We argued in the proof of Claim 3 that, in each of these cases, $s$ changes $p.next$ from $r$ to point to a new Node, and that this is the only change that it makes. Let $I$ be the invocation of INSERT or DELETE in which $s$ occurs.

Suppose $s$ occurs at line 34. We argued in the proof of Claim 3 that, at all times between when the LLX($r$) at line 30 and $s$ are linearized, $p.next$ points to $r$ and $r.next$ points to some Node $rnext$. Therefore, new.$key = r.key$ and new.$next$ points to $rnext$. We show $r$ is a Node (and not the head entry point), and $r.key \neq \infty$. Since $r.next$ points to a Node $rnext \neq \text{Nil}$, $r \neq \text{Nil}$ and $r.key \neq \infty$ (by the inductive hypothesis). Similarly, since $p.next$ points to $r$, either $r = \text{Nil}$ or $r$ is a Node, so we are done. Since $r.next$ points to $rnext$ just before $s$ is linearized, setting new.$next$ to point to $rnext$ does not violate the inductive hypothesis. Since $p.next$ points to $r$, the inductive hypothesis implies that either $p$ is the head entry point or $p.key < r.key$. Clearly, setting $p.next$ to point to new does not violate the inductive hypothesis in either case.

Suppose $s$ occurs at line 24. Then, new.$key = key$ and new.$next$ points to $r$. Before $s$, $I$ invokes SEARCH($key$) at line 16, and then sees $key \neq r.key$ at line 17. By inductive Claim 2, this invocation of SEARCH satisfies its post-conditions, which implies that $r$ points to a Node which satisfies $key < r.key$. Since SEARCH($key$) is invoked only when $key < \infty$ (by inspection of the code and preconditions), $key < \infty$. Thus, setting new.$next$ to point to $r$ does not violate the inductive hypothesis. The post conditions of SEARCH also imply that either $p$ is a Node and $p.key < key$ or $p = \text{head}$. Therefore, setting $p.next$ to point to new does not violate the inductive hypothesis.

Suppose $s$ occurs at line 37. We argued in the proof of Claim 3 that, at all times between when the LLX($r$) at line 30 and $s$ are linearized, $p.next$ points to $r$, $r.next$ points to some Node $rnext$ (pointed to by localr.$next$) and $rnext.next$ points to some Node $rnext'$. Thus, new.$key = rnext.key$ and new.$next$ points to $rnext'$. Since $rnext.next$ points to a Node $rnext'$, $rnext'.key < \infty$ (by the inductive hypothesis). Therefore, since $rnext.next$ points to $rnext'$ just before $s$, setting new.$next$ to point to $rnext'$ does not violate the inductive hypothesis. By the inductive hypothesis, either $p$ is the head entry point, or $p.key < r.key < rnext.key = new.key < \infty$. Clearly, setting $p.next$ to point to new does not violate the inductive hypothesis in either case.

Corollary 104. The head entry point always points to a sorted list with strictly increasing keys.

Proof. Immediate from Lemma 103.4.

Observation 105. Lemma 103.3 implies Constraint 98.

We now argue that the multiset algorithm satisfies a constraint placed on the use of LLX and SCX. This constraint is used to guarantee progress for SCX.

Observation 106. Consider any execution that contains a configuration $C$ after which no field of any Data-record changes. There is a total order on all Data-records created during this execution such that, if Data-record $r_1$ appears before Data-record $r_2$ in the sequence $V$ passed to an invocation $S$ of SCX whose linked LLXs begin after $C$, then $r_1 < r_2$.

Proof. Since the LLXs linked to $S$ begin after $C$, it follows immediately from the multiset code that $V$ is a subsequence of nodes in the list. By Corollary 104, they occur in order of strictly increasing keys, so $r_1$ before $r_2$ in $V$ implies $r_1.key < r_2.key$. Thus, we take the total order on keys to be our total order.
We call an invocation of `INSERT` or `DELETE` **effective** if it performs a linearized invocation of SCX (which either returns `True`, or does not terminate). From the code of `INSERT` and `DELETE`, each **effective** invocation of `INSERT` or `DELETE` performs exactly one linearized invocation of SCX, each invocation of `INSERT` that returns is effective, and each invocation of `DELETE` that returns `True` is effective. We linearize each effective invocation of `INSERT` or `DELETE` at its linearized invocation of SCX. The linearization point for an invocation `I` of `DELETE(key, count)` that returns `False` is subtle. Suppose `I` returns `False` after seeing `r.key ≠ key`. Then, we must linearize it at a time when the node `p` and `r` returned by its invocation `I'` of `SEARCH` are both in the data structure and `p.next` points to `r`. By Observation 105, Constraint 98 is satisfied. This means we can apply Lemma 102 to show that there is a time during `I'` when `p` is in the data structure and `p.next = r` (so `r` is also in the data structure). We linearize `I` at the last such time. Now, suppose `I` returns `False` after seeing `r.count < count`. Then, we must linearize it at a time when the node `r` returned by its invocation `I'` of `SEARCH` is in the data structure, and satisfies `r.count < count`. As in the previous case, we can apply Lemma 102 to show that there is a time after the start of `I'`, and at or before when `I` reads a value `v` from `r.count` at line 32, such that `r` is in the data structure and `r.count = v`. We linearize `I` at the last such time. Similarly, we linearize each `GET` at the last time after the start of the `SEARCH` in `GET`, and at or before when the `GET` reads a value `v` from `r.count`, such that `r` is in the data structure and `r.count = v`. Clearly, each operation is linearized during that operation.

**Lemma 108.** At all times `t`, the multiset `σ` of keys in the data structure is equal to the multiset `σ_L` of keys that would result from the atomic execution of the sequence of operations linearized up to time `t`.

**Proof.** We prove this claim by induction on the sequence of steps taken in the execution. Since `next` pointers and `count` fields can be changed only by linearized invocations of SCX (and `key` fields do not change), we need only consider linearized invocations of SCX when reasoning about `σ`. Thus, invocations of `INSERT` and `DELETE` that are not effective cannot change the data structure. Since invocations of `GET` do not invoke SCX, they cannot change the data structure. Therefore, we need only consider effective invocations of `INSERT` and `DELETE` when reasoning about `σ_L`. Since each effective invocation of `INSERT` or `DELETE` is linearized at its linearized invocation of SCX, the steps that can affect `σ` and `σ_L` are exactly the same. **Base case.** Before any linearized SCX has occurred, no `next` pointer has been changed. Thus, the data structure is in its initial configuration, which implies `σ = ∅`. Since no effective invocation of `INSERT` or `DELETE` has been linearized, `σ_L = ∅`. **Inductive step.** Let `s` be a linearized invocation of `S` of SCX(`V`, `R`, `field`, `new`). `I` be the (effective) invocation of `INSERT` or `DELETE` that performs `S`, and `p` and `r` and `rnext` refer to the local variables of `I`. Suppose `σ = σ_L` before `S`. Let `σ'` denote `σ` after `S`, and `σ_L'` denote `σ_L` after `S`. We prove `σ' = σ_L'`. Suppose `S` is performed at line 20. Then, `I` is an invocation of `INSERT(key, count)`, and `σ_L' = σ_L + {count copies of key}`. By Lemma 103.3, no Data-record is added or removed by `S`. Before `I` performs `S`, `I` performs an invocation `L` of `LLX(r)` linked to `S` at line 18. Since `S` is linearized, no mutable field of `r` changes between when `L` and `S` are linearized. Therefore, the value `localr.count` that `L` reads from `r.count` is equal to the value of `r.count` at all times between when `L` and `S` are linearized, and line 20 implies that `S` changes `r.count` from `localr.count` to `localr.count + count`. Since `S` is linearized, Lemma 99 implies that `r` must be in the data structure just before `S` is linearized. By Lemma 103.4, `r` is the only Node in the data structure with key `key`, so `σ` contains exactly `v` copies of `key` just before `S` is linearized. Since this is the only change made by `S`, `σ' = σ + {count copies of key}`, and the inductive hypothesis implies `σ' = σ_L'`. Suppose `S` is performed at line 24. Then, `I` is an invocation of `INSERT(key, count)`, and `σ_L' = σ_L + {count copies of key}`. By Lemma 103.3, no Data-record is removed by `S`, and only `new` is added by `S`. From the code of `INSERT`, `new.key = key` and `new.count = count`. Therefore, `σ' = σ + {count copies of key}`, and the inductive hypothesis implies `σ' = σ_L'`. Suppose `S` is performed at line 34. Then, `I` is an invocation of `DELETE(key, count)`. Before `I` performs `S`, `I` performs an invocation `L` of `LLX(r)` linked to `S` at line 30. Since `S` is linearized, no mutable field of `r` changes between when `L` and `S` are linearized. Thus, the value `localr.count` that `L` reads from `r.count` is equal to the value of `r.count` at all times between when `L` and `S` are linearized. This implies that `I` sees `r.key = key` and `r.count ≥ count` at line 32. By Lemma 103.3, `r` is the only Data-record removed by `S`, and `new` is the only Data-record added by `S`. By Definition 97, `r` must be in the data structure just before `S` is linearized. By Lemma 103.4, `r` is the only Node in the data structure with key `key`. Hence, `σ` contains exactly `localr.count` copies of `key` just before `S` is linearized. From the code of `DELETE`, `new.key = r.key` and `new.count = localr.count - count`. Therefore, `σ' = σ - {count copies of key}`. By the inductive hypothesis, `σ = σ_L`. Thus, there are `localr.count ≥ count` copies of `key` in `σ_L`. Therefore, if `I` is performed atomically at its linearization point, it will enter the `if`-block at line 33, so `σ_L' = σ_L - {count copies of key} = σ'`. Suppose `S` is performed at line 37. Then, `I` is an invocation of `DELETE(key, count)`. Before `I` performs `S`, `I` performs an invocation `L` of `LLX(r)` linked to `S` at line 30. Since `S` is linearized, no mutable field of `r` changes between when `L` and `S` are linearized. Thus, the value `localr.count` that `L` reads from `r.count` is equal to the value of `r.count` at all times between when `L` and `S` are linearized. This implies that `I` sees `r.key = key` and `r.count ≥ count` at line 32, and `count ≥ r.count` at line 33. Hence, `r.count = count` at all times between when `L` and `S` are linearized. Let `rnext` be the Node pointed to by `I`'s local variable `localr.next`. (We know `rnext` is a Node, and not `NIL`, from `r.key = key < ∞` and Lemma 103.4.) After `L`, `I` performs an invocation `L'` of `LLX(rnext)` linked to `S` at line 36. By the same argument as for `r.count`, the value `v` that `L'` reads from `rnext.count` is equal to the value of `rnext.count` at all times between when `L'` and `S` are linearized. By Lemma 103.3, `r` and `rnext` are the only Data-records removed by `S`, and `new` is the only Data-record added by `S`. By Definition 97, `r` and `rnext` must be in the data structure just before `S`. By Lemma 103.4, `r` is the only Node in the data structure with key `key`, and `rnext` is the only Node in the data structure with its key. Hence, `σ` contains exactly `r.count = count` copies of `key`, and exactly `v` copies of `rnext.key`. From the code of `DELETE`, `new.key = rnext.key` and `new.count = rnext.count = v`. 
Therefore, \( \sigma' = \sigma - \{ \text{count copies of key} \} \) By the inductive hypothesis, \( \sigma = \sigma_L \). Thus, there are exactly \( \text{count} \) copies of \( \text{key} \) just before \( I \) in the linearized execution. From the code of \( \text{DELETE} \), in the linearized execution, \( I \) will enter the else block at line 35, so \( \sigma'_L = \sigma_L - \{ \text{count copies of key} \} = \sigma' \).

**Lemma 109.** Each invocation of \( \text{GET(key)} \) that terminates returns the number of occurrences of \( \text{key} \) in the data structure just before it is linearized.

**Proof.** Consider any invocation \( I \) of \( \text{GET(key)} \). Let \( I' \) be the invocation of \( \text{SEARCH(key)} \) performed by \( \text{GET(key)} \), and \( p \) and \( r \) refer to the local variables of \( I' \). By Lemma 103.2, \( I' \) satisfies its postcondition, which means that \( \text{key} \leq r\.key \), and either \( p\.key < \text{key} \) or \( p = \text{head} \). We proceed by cases. Suppose \( \text{key} = r\.key \). Then, after \( I' \), \( I \) reads a value \( v \) from \( r\.count \) and returns \( v \). By Observation 105, Constraint 98 is satisfied. By Lemma 102, there is a time after the start of \( I' \), and at or before when \( I \) reads \( r\.count \), such that \( r \) is in the data structure and \( r\.count = v \). \( I \) is linearized at the last such time. By Corollary 104, \( r \) is the only Data-record in the list that contains \( \text{key} \). Suppose that either \( \text{key} < r\.key \) and \( p = \text{head} \), or \( \text{key} < r\.key \) and \( p\.key < \text{key} \). Then, \( I \) returns zero. By Lemma 102, at sometime during \( I' \), \( p \) was in the data structure and \( p\.next \) pointed to \( r \). \( I \) is linearized at the last such time. By Corollary 104, the data structure contains no occurrences of \( \text{key} \) when \( I \) is linearized.

**Lemma 110.** Each invocation \( I \) of \( \text{DELETE(key,count)} \) that terminates returns \( \text{True} \) if the data structure contains at least \( \text{count} \) occurrences of \( \text{key} \) just before \( I \) is linearized, and \( \text{False} \) otherwise.

**Proof.** **Case I:** \( I \) returns \( \text{False} \). In this case, \( I \) satisfies \( \text{key} \neq r\.key \) or \( l\.localr\.count < \text{count} \) at line 32. Suppose \( \text{key} \neq r\.key \). Then, by the postcondition of \( \text{SEARCH} \), \( \text{key} < r\.key \), and either \( p\.key < \text{key} \) or \( p = \text{head} \). By Observation 105, Constraint 98 is satisfied. By Lemma 102, there is a time during \( I' \), \( p \) was in the data structure and \( p\.next \) pointed to \( r \). \( I \) is linearized at the last such time. Corollary 104 implies that there are no occurrences of \( \text{key} \) in the data structure when \( I \) is linearized. By the precondition of \( \text{DELETE} \), \( \text{count} > 0 \), so the claim is satisfied.

Now, suppose \( l\.localr\.count < \text{count} \) at line 32. By Lemma 102, there is a time after the start of \( I' \), and before \( I \)'s \( \text{LLX(r)} \) reads \( l\.localr\.count \) from \( r\.count \), such that \( r \) is in the data structure and \( r\.count = l\.localr\.count \). \( I \) is linearized at the last such time. By Corollary 104, \( r \) is the only Data-record in the list that contains \( \text{key} \), so there are \( r\.count < \text{count} \) occurrences of \( \text{key} \) = \( \text{key} \) in the data structure when \( I \) is linearized.

**Case II:** \( I \) returns \( \text{True} \). In this case, \( I \) satisfies \( \text{key} = r\.key \) and \( l\.localr\.count \geq \text{count} \) at line 32, and \( I \) is linearized at an invocation \( S \) of \( \text{SCX} \) at line 34 or 37. In each case, Lemma 103.3 implies that \( r \) is removed by \( S \), so \( r \) is in the data structure just before \( S \) is linearized. Hence, \( r \) is in the data structure just before \( I \) is linearized. Before \( I \) performs \( S \), \( I \) performs an invocation \( L \) of \( \text{LLX(r)} \) linked to \( S \) at line 30 that reads \( l\.localr\.count \) from \( r\.count \). Since \( S \) is linearized, no mutable field of \( r \) changes between when \( L \) and \( S \) are linearized. Therefore, the value of \( l\.localr\.count \) is equal to the value of \( r\.count \) at all times between when \( L \) and \( S \) are linearized. Thus, just before \( I \) is linearized, \( r \) is in the data structure and \( r\.count \geq \text{count} \). Finally, Corollary 104 implies that \( r \) is the only Data-record in the list that contains \( \text{key} \), so the claim holds.

We now prove that our algorithm satisfies an assumption that we made in the paper.

**Lemma 111.** No process performs more than one invocation of \( \text{LLX(r)} \) that returns \( \text{FINALIZED} \), for any Data-record \( r \).

**Proof.** Let \( r \) be a Data-record. Suppose, to derive a contradiction, that a process \( p \) performs two invocations \( L \) and \( L' \) of \( \text{LLX(r)} \) that return \( \text{FINALIZED} \). Without loss of generality, let \( L \) occur before \( L' \). From the code of \( \text{INSERT} \) and \( \text{DELETE} \), \( p \) must perform an invocation of \( \text{SEARCH} \), \( L \), another invocation of \( I \) of \( \text{SEARCH} \), and then \( L' \). Since \( L \) returns \( \text{FINALIZED} \), it is linearized after an invocation \( S \) of \( \text{SCX(V,R,.fd,new)} \) with \( r \in R \). By Lemma 103.3, \( r \) is removed from the data structure by \( S \). We now show that \( r \) cannot be added back into the data structure by any subsequent invocation of \( \text{SCX} \). From the code of \( \text{INSERT} \) and \( \text{DELETE} \), each invocation of \( \text{SCX(V',R',fd,new')} \) that changes a \( \text{next} \) pointer is passed a newly created \( \text{Node} \), that is not known to any other process, as its new' argument. This implies that new' is not initiated, and cannot have previously been removed from the data structure. Therefore, \( r \) is not in the data structure at any point during \( I \). By Observation 105, Constraint 98 is satisfied. By Lemma 102, \( r \) is in the data structure at some point during \( I \), which is a contradiction.

**Lemma 112.** If operations \( (\text{INSERT, DELETE and GET}) \) are invoked infinitely often, then operations complete infinitely often.

**Proof.** Suppose, to derive a contradiction, that operations are invoked infinitely often but, after some time \( t \), no operation completes. If \( \text{SCXs} \) are performed infinitely often, then they will succeed infinitely often and, hence, operations will succeed infinitely often. Thus, there must be some time \( t' > t \) after which no \( \text{SCX} \) is performed. Then, after \( t' \), the data structure does not change, and only a finite number of nodes with keys different from \( \infty \) are ever added to the data structure. Consider an invocation \( I \) of \( \text{SEARCH(key)} \) that is executing after \( t' \). Each time \( I \) performs line 12, it reads a Node \( \text{n.next} \) from \( \text{n.next} \), and \( \text{n.next.key} > \text{r.key} \). Therefore, by Corollary 104, \( I \) will eventually see \( r\.key = \infty \) at line 10. This implies that every invocation of \( \text{GET} \) eventually completes. Therefore, \( \text{INSERT and DELETE} \) must be invoked infinitely often after \( t' \). From the code of \( \text{INSERT (DELETE),} \) in each iteration of the while loop, a \( \text{SEARCH} \) is performed, followed by a sequence of \( \text{LLXs} \). If these \( \text{LLXs} \) all return values different from \( \text{FAIL or FINALIZED} \), then an invocation of \( \text{SCX} \) is performed. Since every invocation of \( \text{SEARCH} \) eventually completes, Definition 95 implies that invocations of \( \text{SCX} \) are set up infinitely often. Thus, invocations of \( \text{SCX} \) succeed infinitely often. From the code of \( \text{INSERT and DELETE}, \) after performing a successful invocation of \( \text{SCX} \), an invocation of \( \text{INSERT or DELETE} \) will immediately return.