Lock-Free Reference Counting

David L. Detlefs, Paul A. Martin, Mark Moir, Guy L. Steele Jr.

Sun Microsystem Laboratories, 1 Network Drive, Burlington, MA 01803

Received: 1 January 2002 / Revised version: 15 July 2002

Summary. Assuming the existence of garbage collection makes it easier to design implementations of dynamic-sized concurrent data structures. However, this assumption limits their applicability. We present a methodology that, for a significant class of data structures, allows designers to first tackle the easier problem of designing a garbage-collection-dependent implementation, and then apply our methodology to achieve a garbage-collection-independent one. Our methodology is based on the well-known reference counting technique, and employs the double compare-and-swap operation.

1 Introduction

We present a methodology that can be used to simplify the design of dynamic-sized nonblocking shared data structures. The considerable amount of research attention that has been paid in recent years to the implementation of nonblocking data structures has mostly focused on lock-free and wait-free implementations. An implementation is wait-free if it guarantees that after a finite number of steps of any operation, the operation completes (regardless of the timing behaviour of threads executing other operations concurrently). An implementation is lock-free if it guarantees that after a finite number of steps of any operation, *some* operation completes. Both properties preclude the use of mutually exclusive locks for synchronization because if some thread is delayed while holding a lock, then no other operation that requires the lock can complete — no matter how many steps it takes — until the lock holder releases the lock. Lock-free programming is increasingly important for overcoming this and other problems associated with locking, including performance bottlenecks, susceptibility to delays and failures, design complications, and, in real-time systems, priority inversion.

Lock-freedom is weaker than wait-freedom, as it guarantees progress only on a system-wide basis, while wait-freedom guarantees per-operation progress. Research experience shows, however, that the stronger wait-freedom property is much more difficult to achieve, and usually results in more costly implementations. Furthermore, lock-freedom is likely to be sufficient in practice because in a well-designed system, contention should be low. In this paper, we concentrate on the design of lock-free data structures, and specifically *dynamic-sized* ones, that is, data structures whose size can grow and shrink over time.

It is well known that garbage collection (GC) can simplify the design of sequential implementations of dynamicsized data structures. Furthermore, in addition to providing storage management benefits, GC can also significantly simplify various synchronization issues in the design of concurrent data structures [3,15,22]. Unfortunately, however, simply designing implementations for garbage-collected environments is not the whole solution. First, not all programming environments support GC. Second, almost all of those that do employ excessive synchronization, such as locking and/or stop-the-world mechanisms, which brings into question their scalability. Finally, an obvious restriction of particular interest to our group is that GC-dependent implementations cannot be used in the implementation of the garbage collector itself!

The goal of this work, therefore, is to allow programmers to exploit the advantages of GC in designing their lock-free data structure implementations, while avoiding its drawbacks. To this end, we provide a methodology that allows programmers to first solve the easier problem of designing a GC-dependent implementation, and to then apply our methodology in order to achieve a GC-independent one.

We have designed our methodology to preserve lock-freedom. That is, if the GC-dependent implementation is lock-free so too will be the GC-independent one derived using our methodology. Two clarifications are needed here. First, some readers may be concerned about our reference to lock-free, GC-dependent implementations,

given that most garbage collectors stop threads from executing while they perform their work. However, this does not mean that the *data structure implementation* is itself not lock-free: the definition if lock-freedom does not require progress if the system (say, the GC or the OS) prevents threads from taking any steps.

Second, we say that we achieve lock-free, GC-independer implementations of dynamic-sized data structure by applying our methodology, but we do not specify how objects are created and destroyed; usually malloc and free are not lock-free, so implementations based on them are also not lock-free. However, most production-quality memory allocators do attempt to avoid contention for locks — for example by maintaining separate allocation buffers for each thread — and therefore avoid most of the problems associated with locks. In fact, our colleagues Dice and Garthwaite have recently proposed a new memory allocator [5] that is designed to be "multiprocessor-aware" in order to improve locality and reduce synchronization overhead. As a result, their allocator resorts to locking only very occasionally, and therefore scales very well. While their allocator does still use locks, variations on straightforward and well-known nonblocking techniques could be applied to make it completely nonblocking.

An important feature of our methodology — and data structure implementations based on it — is that it allows the memory consumption of the implementation to grow and shrink over time, without imposing any restrictions on the underlying memory allocation mechanisms. In contrast, lock-free implementations of dynamic data structures often either require maintenance of a special "freelist", whose storage cannot in general be reused for other purposes (e.g. [19,25]), or require special system support such as type stable memory [7].

Our methodology is based on the well-known garbage collection technique of reference counting. We refer to our methodology as LFRC (Lock-Free Reference Counting). In this approach, each object contains a count of the number of pointers that point to it, and can be freed if and only if this count is zero. The reference counting approach to memory management has the limitation that it does not detect cycles in garbage (imagine a ring of unreachable objects in which each has a pointer to the next — all reference counts are at least one, so none of the objects are ever identified as garbage). LFRC inherits this limitation. In many cases, either cycles do not exist in garbage, or the implementation can be easily modified to eliminate this possibility. In other cases, it is more difficult or even impossible to do so; in such cases our methodology cannot guarantee that all garbage will be reclaimed.

In order to maintain accurate reference counts, we would like to be able to atomically create a pointer to an object and increment that object's reference count, and to atomically destroy a pointer to an object and decrement its reference count. This way, by freeing an object when and only when its reference count becomes zero, we can ensure that objects are not freed prematurely, but that they are eventually freed when no pointers to the object remain.

The main difficulty that arises in the above-described approach is the need to atomically modify two separate memory locations: a pointer and the reference count of the object to which it points. This can be achieved either by using hardware synchronization primitives to enforce this atomicity, or by using lock-free software mechanisms to achieve the appearance of atomicity. Work on the latter approach has yielded various lock-free and waitfree implementations of multi-variable synchronization operations (e.g. [20,23]). However, all of these results are complicated, introduce significant overhead, and/or work only on locations in statically-allocated memory. rendering them unsuitable for our goal of supporting dynamic-sized nonblocking data structures. Therefore, in this paper, we explore the former approach. In particular, we assume the availability of a double compareand-swap (DCAS) instruction that can atomically access two independently-chosen memory locations. (DCAS is precisely defined in Section 2.) While such an instruction is not widely available, it has been implemented in hardware in the past (e.g. [21]). Furthermore, it is important to investigate the role of hardware synchronization mechanisms in determining the feasibility of scalable nonblocking synchronization. Thus it is appropriate to study what can be achieved with alternative hardware mechanisms.

Even DCAS does not allow us to maintain reference counts that are accurate at all times. For example, if a pointer in shared memory points to an object v, and we change the pointer to point to another object w, then we have to atomically modify the pointer, increment w's reference count, and decrement v's reference count. However, it turns out that a weaker requirement on the reference counts suffices, and that this requirement can be achieved using DCAS. This weakening is based on the observation that reference counts do not always need to be accurate. The important requirements are that if the number of pointers to an object is non-zero, then so too is its reference count, and that if the number of pointers is zero, then the reference count eventually becomes zero. These two requirements respectively guarantee that an object is never freed prematurely, and that the reference count of an object that has no pointers to it eventually become zero, so that it can be freed.¹

These observations imply that it is safe for a thread to increment an object's reference count *before* creating a new pointer to it, provided that the thread eventually either creates the pointer, or decrements the reference count to compensate for the previous increment. This is the approach used by our methodology.

The observation that a weaker property for reference counts is sufficient entices us to consider using a single-location compare-and-swap (CAS) instruction, which is widely available, to maintain reference counts. However, when we load a pointer from a shared memory location, we need to increment the reference count of the object to which the loaded value points. If we can access this ref-

¹ To be precise, we should point out that it is possible for garbage to exist and to never be freed in the case where a thread fails permanently.

erence count only with a single-location CAS, then there is a risk that the object will be freed before we increment the reference count, and that the subsequent attempt to increment the reference count will corrupt memory that has been freed, and potentially reallocated for another purpose. By using DCAS, we can increment the reference count while atomically ensuring that a pointer still exists to this object. While this scenario explains why a naive approach to using CAS for maintaining reference counts does not work, it is by no means a formal impossibility argument. In fact, our subsequent work on constructing such an argument resulted in an effective CAS-based approach for handling memory management in nonblocking dynamic-sized shared data structures [8– 10. This development adds more weight to our argument that research based on synchronization primitives that are not widely available is appropriate and important.

To demonstrate the utility of the LFRC methodology, we show how it can be used to convert a GCdependent algorithm published recently in [3] into a GCindependent one. The algorithm presented in [3] is a lockfree implementation of a double-ended queue (hereafter "deque") [14]. The design of the implementation in [3] was simplified by the assumption that it would operate in an environment that provides GC because GC can give us a free solution to the so-called ABA problem ("free" in the sense that we do not have to solve this problem as part of the implementation). This problem arises when we fail to detect that a value changes and then changes back to its original value. For example if a CAS or DCAS operation is about to operate on a pointer, and the object to which it points is freed and then reallocated, then it is possible for the CAS or DCAS to succeed even though it should fail. Thus, this situation must be prevented by ensuring that objects are not freed while threads have pointers to them. Garbage collectors can determine this, for example by stopping a thread and inspecting its registers and stack. However, in the absence of GC, the implementation itself must deal with this tricky problem. It was while working on this specific problem that we realised that we could achieve a more generally-applicable methodology.

While we believe that our methodology will be useful for a variety of concurrent data structures, it is not completely automatic, nor is it universally applicable, primarily for the following two reasons: our methodology is applicable to implementations that manipulate pointers using only a predefined set of operations, and also (as explained earlier) to implementations that ensure there are no cycles in garbage. We define the class of implementations to which our methodology is applicable more precisely in the following section.

The remainder of this paper is organized as follows. We define DCAS and state system assumptions in Section 2. In Section 3, we describe the operations supported by LFRC, and in Section 4, we describe in general how they can be applied. Then, in Section 5, we demonstrate a specific example by using LFRC to convert the GC-dependent lock-free deque implementation from [3] into a GC-independent one. In Section 6, we present lock-free implementations for the LFRC operations. Related work

is briefly discussed in Section 7, and concluding remarks appear in Section 8. A correctness proof is presented in an appendix.

2 Definitions and Assumptions

We assume a sequentially-consistent [17] shared memory multiprocessor system that supports atomic load, store, compare-and-swap (CAS) and double compare-and-swap (DCAS) operations.

The DCAS operation accepts six parameters: two addresses A0 and A1, two old values old0 and old1, and two new values new0 and new1. It atomically compares (for equality) the value stored at address A0 with old0 and the value stored at address A1 with old1 and, if the comparisons both succeed, stores new0 at address A0 and new1 at address A1, and returns true; if either comparison fails (i.e., one of the old values differs from the contents of the corresponding address), then the DCAS operation returns false and does not modify shared memory. CAS is the obvious simplification of DCAS that accepts only one address, one old value, and one new value.

3 LFRC

In this section, we describe the class of algorithms to which LFRC is applicable and the LFRC operations.

3.1 Applicability

The LFRC operations support loading, storing, copying, and destroying pointers, as well as modifying them using CAS and DCAS. We present a methodology for transforming any garbage-collection-dependent concurrent data structure implementation that satisfies the two criteria below into an equivalent implementation that does not depend on GC.

LFRC Compliance The implementation does not access or manipulate pointers other than by loading, storing, copying (including passing as parameters), or destroying them, or by modifying them using CAS and/or DCAS. (For example, this precludes the use of pointer arithmetic.) Furthermore, all pointer variables used in the implementation are statically-allocated shared variables, fields of dynamically-allocated objects, local variables or parameters, or private global variables. (For example, this precludes the application from modifying pointer variables that are stored on its stack below the call frame of the currently-executing procedure.)

Cycle-Free Garbage There are no pointer cycles in garbage. (Cycles may exist in the data structure, but not amongst objects that have been removed and should be freed.)

The set of operations allowed by the LFRC Compliance criterion above seems to be sufficient to support

a wide range of concurrent data structure implementations. Given the general principles demonstrated in this paper, it should be straightforward to extend our methodology to support other operations such as load-linked and store-conditional.

The Cycle-Free Garbage criterion will clearly disqualify some implementations, but we believe that it is easily achievable for a wide range of concurrent data structure implementations. In many cases, the criterion will be satisfied by the natural implementation; in others some effort may be required to eliminate garbage cycles. In the example presented in Section 5, the original implementation did allow the possibility of cycles within garbage, but we overcame this problem with a straightforward modification. We have several other candidate implementations to which we believe this technique will be applicable.

3.2 LFRC Operations

LFRC supports the following operations for accessing and manipulating pointers. As stated above, we assume that pointers are accessed *only* by means of these operations.

- LFRCLoad(p,A) A is a pointer to a shared memory location that contains a pointer, and p is a pointer to a local pointer variable. The effect is to load the value from the location pointed to by A into the variable pointed to by p.
- LFRCStore(A,v) A is a pointer to a shared memory location that contains a pointer, and v is a pointer value to be stored in this location.
- LFRCStoreAlloc(A, new sometype) A is a pointer to a shared memory location that contains a pointer to sometype. Using LFRCStoreAlloc() is more convenient than explicitly saving the pointer returned by new so that it can be immediately passed to LFRCDestroy().
- LFRCCopy(p,v) p is a pointer to a local pointer variable and v is a pointer value to be copied to the variable pointed to by p.
- LFRCPass(p) p is a local pointer variable being passed by value to an application (non-LFRC) procedure or function.
- LFRCDestroy(v) v is the value of a local pointer variable that will never be used again (for example because it is about to go out of scope because of a return from a function).
- LFRCCAS (A0,old0,new0) is the obvious simplification of LFRCDCAS, described next.
- LFRCDCAS(A0,A1,old0,old1,new0,new1) A0 and A1 are pointers to shared memory locations that contain pointers, and old0, old1, new0, and new1 are pointer values. The effect is to atomically compare (for equality) the contents of the location pointed to by A0 with

oldO and the contents of the location pointed to by A1 with old1, and to change the contents of the locations pointed to by A0 and A1 to newO and new1, respectively, returning true, if the comparisons both succeed; if either comparison fails (i.e., one of the old values differs from the contents of the corresponding address), then the contents of the locations pointed to by AO and A1 are left unchanged, and LFRCDCAS returns false.

4 LFRC Methodology

In this section we describe the steps of our methodology for transforming a GC-dependent implementation into a GC-independent one. All of these steps except step 3 can be completely automated in an object-oriented language such as C++ using "smart pointer" techniques like those described in [2]. However, presenting these steps explicitly is clearer, and allows our methodology to be applied in non-object-oriented languages too. The steps are as follows:

- 1. Add reference counts: Add a field rc to each object type to be used by the implementation. This field should be set to 1 in a newly-created object (in our C++ implementation, this is achieved through object constructors).
- 2. Provide an LFRCDestroy() function: This function should accept an object pointer v as an argument. If v is null, then the function should simply return; otherwise it should atomically decrement v rc. If v rc becomes zero as a result, LFRCDestroy() should recursively call itself with each pointer in the object, and then free the object. An example is provided in Section 6, and we provide a function add_to_rc() for the atomic decrement of the rc field, so writing this function is trivial. (We require the LFRCDestroy() function to be supplied only because this is the most convenient language-independent way to iterate over all pointers in an object.)
- **3. Ensure no garbage cycles:** Observe that the reference counts of nodes in a garbage cycle will remain non-zero forever. Therefore, to ensure that all garbage is collected, we must ensure that the implementation does not result in cycles among garbage objects. (Failing to achieve this will result in the memory on and reachable from the cycle being lost, but will not affect the correctness of the implemented data structure.) This step may be non-trivial or even impossible for some concurrent data structure implementations. If this property cannot be achieved for a particular data structure, then it is not a candidate for applying our methodology.
- 4. Produce correctly-typed LFRC operations: We have provided code for the LFRC operations to be used in the example implementation presented in the next section. In this implementation, there is only one type of pointer; for simplicity and clarity, we have explicitly designed the LFRC operations for this type. For similarly simple implementations, this step can be achieved by a simple search and replace. In algorithms that use

Original Code	Replacement Code
x0 = *A0;	LFRCLoad(&x0,A0);
*A0 = x0;	LFRCStore(A0,x0);
*AO = new sometype;	LFRCStoreAlloc(A0, new sometype);
*AO = *A1;	<pre>{ ObjectType *x = null; LFRCLoad(&x,A1); LFRCStore(A0,x); LFRCDestroy(x); }</pre>
x0 = x1;	LFRCCopy(&x0,x1);
proc(x0);	<pre>proc(LFRCPass(x0));</pre>
CAS(A0,old0,new0)	LFRCCAS(A0,old0,new0)
DCAS(A0,A1,old0,old1,new0,new1)	LFRCDCAS(A0,A1,old0,old1,new0,new1)

Table 1. Step 5: Replacing pointer accesses in original code with LFRC counterparts. A0 and A1 are pointers to shared pointer variables, and x0, x1, old0, old1, new0, new1 are local pointer variables or parameters or private global pointer variables.

multiple pointer types, either the operations will need to be duplicated for different pointer types, or the code for the operations will need to be unified to accept different pointer types. For example, this step could be eliminated completely by arranging for the rc field to be at the same offset in all objects, and by using void pointers instead of specifically-typed ones.

5. Replace pointer operations: Replace each pointer operation with its LFRC counterpart as shown in Table 1

6. Management of local pointer variables and parameters: Whenever a thread loses a pointer (for example when a function that has local pointer variables or parameters returns, so its local variables and parameters go out of scope), it first calls LFRCDestroy() with this pointer. Also, all pointer variables must be initialized to null before being used with any of the LFRC operations. Thus, all pointers in a newly-allocated object should be initialized to null before the object is made visible to other threads by invoking an LFRC procedure with a pointer to the object. It is also important to explicitly destroy pointers contained in a statically-allocated object (by using LFRCStore to replace them with null) before destroying that object; see Section 5 for an example. (This is necessary because LFRCDestroy() is never invoked for statically-allocated objects, and therefore will not be recursively invoked on the pointer fields of such objects.)

5 An Example: Snark

In this section we show how to use our methodology to construct a GC-independent implementation of a concurrent deque, based on the GC-dependent one presented in [3] and affectionately known as "Snark". Below we describe some of the relevant (to the application of our methodology) features of this algorithm, in preparation for explaining the application of the LFRC methodology to this algorithm; readers interested in the details are referred to [3].

The C++ class declarations² and code for one operation — pushRight() — of the original Snark algorithm are shown on the left side of Figure 1. Snark represents a deque as a doubly-linked list. The nodes of the doublylinked list are called SNodes (lines 1..2); they contain L and R pointers, which point to their left and right neighbours, respectively. A Snark deque consists of two "hat" pointers (LeftHat and RightHat), which respectively point to the leftmost and rightmost nodes in a non-empty deque, and a special node Dummy, which is used as a sentinel node at one or both ends of the deque (line 3). The Snark constructor (lines 4..9) allocates an SNode for Dummy, makes the L and R pointers of Dummy point to Dummy itself, and makes LeftHat and RightHat both point to Dummy. This is one representation of the empty deque in this implementation. The self-pointers in the L and R pointers of the dummy node are used to distinguish this node from nodes that are currently part of the deque. Some pop operations leave a previous deque node as a sentinel node; in this case, a pointer in this node is also directed toward the node itself to identify the node as a sentinel. Pointers are accessed only by load, store, and DCAS operations.

The right side of Figure 1 shows the GC-independent code we derived from the original Snark algorithm by using the LFRC methodology. Below we briefly explain the differences between the two implementations, and explain how we applied the steps of the LFRC methodology to the first in order to acquire the second. (There are various obvious optimizations that could be applied to the modified algorithm, for example eliding calls to LFRCDestroy() with arguments that we know are null; we have not applied such optimizations in order to preserve the exact results of the transformation dictated by the LFRC methodology.)

Step 1 We added a reference count field (rc) to the SNode object (line 31); the SNode constructor sets it to 1

² For clarity and brevity, we have omitted uninteresting details such as access modifiers.

```
class SNode {
                                                               class SNode {
    class SNode *L, *R; valtype V;
                                                                   class SNode *L, *R; valtype V; long rc;
                                                                   SNode(): L(null), R(null), rc(1) \{\};
    SNode() \{\};
                                                               };
};
class Snark {
                                                               class Snark {
    SNode *Dummy, *LeftHat, *RightHat;
                                                                   SNode *Dummy, *LeftHat, *RightHat;
4
    Snark() {
                                                                   Snark(): Dummy(null),
                                                                                     LeftHat(null), RightHat(null) {
5
        Dummy = new SNode;
                                                               35
                                                                       LFRCStoreAlloc(&Dummy, new SNode):
6
        Dummy \rightarrow L = Dummy;
                                                               36
                                                                       LFRCStore(&Dummy→L, null);
7
        Dummy \rightarrow R = Dummy;
                                                               37
                                                                       LFRCStore(&Dummy→R, null):
        LeftHat = Dummy;
                                                               38
                                                                       LFRCStore(&LeftHat, Dummy);
8
        RightHat = Dummy;
                                                                       LFRCStore(&RightHat, Dummy);
9
                                                               39
    };
                                                                    ~Snark() {
                                                               40
                                                               41
                                                                       while (popLeft()!=EMPTYval);
                                                               42
                                                                       LFRCStore(&Dummy, null);
                                                               43
                                                                       LFRCStore(&LeftHat, null);
                                                               44
                                                                       LFRCStore(&RightHat, null);
                                                                   };
10 valtype pushRight(valtype v);
                                                                   valtype pushRight(valtype v);
                                                               45
11
    valtype pushLeft(valtype v);
                                                               46
                                                                   valtype pushLeft(valtype v);
12
    valtype popRight();
                                                               47
                                                                   valtype popRight();
   valtype popLeft();
                                                                   valtype popLeft();
13
                                                               48
};
                                                               };
valtype Snark::pushRight(valtype v) {
                                                               valtype Snark::pushRight(valtype v) {
14 SNode *nd = new SNode;
                                                                   SNode *nd = new SNode;
    SNode *rh, *rhR, *lh;
                                                                   SNode *rh = null, *rhR = null, *lh = null;
16
   if (nd == null)
                                                               51
                                                                   if (nd == null) {
                                                                       LFRCDestroy(rhR, nd, rh, lh);
                                                               52
17
        return FULLval;
                                                               53
                                                                       return FULLval;
    nd \rightarrow R = Dummy;
                                                                   LFRCStore(&nd→R, Dummy);
18
    nd \rightarrow V = v;
                                                                   nd \rightarrow V = v;
19
20
    while (true) {
                                                               56
                                                                   while (true) {
        rh = RightHat;
                                                                       LFRCLoad(&rh, &RightHat);
21
                                                               57
22
        rhR = rh \rightarrow R:
                                                               58
                                                                       LFRCLoad(&rhR, &rh\rightarrowR);
23
        if (rhR == rh) {
                                                               59
                                                                       if (rhR == null) {
24
           nd \rightarrow L = Dummy;
                                                               60
                                                                          LFRCStore(\&nd \rightarrow L, Dummy);
25
                                                               61
                                                                          LFRCLoad(&lh, &LeftHat);
           lh = LeftHat;
26
           if (DCAS(&RightHat, &LeftHat, rh, lh, nd, nd))
                                                               62
                                                                          if (LFRCDCAS(&RightHat, &LeftHat,
                                                                                                   rh, lh, nd, nd)) {
                                                               63
                                                                              LFRCDestroy(rhR, nd, rh, lh);
27
              return OKval;
                                                               64
                                                                              return OKval;
                                                                       } else {
        } else {
           nd \rightarrow L = rh;
                                                               65
                                                                          LFRCStore(\&nd{\rightarrow}L,\,rh);
28
29
           if (DCAS(\&RightHat, \&rh \rightarrow R, rh, rhR, nd, nd))
                                                               66
                                                                          if (LFRCDCAS(&RightHat, &rh→R,
                                                                                                 rh, rhR, nd, nd)) {
                                                               67
                                                                              LFRCDestroy(rhR, nd, rh, lh);
               return OKval;
                                                                              return OKval;
30
                                                               68
                                                                          }
                                                                       }
                                                                   }
}
                                                               }
```

Fig. 1. Class definitions and pushRight code for GC-dependent Snark (left) and transformed class definitions and pushRight code for GC-independent Snark using LFRC (right). A call to LFRCDestroy() with multiple arguments is shorthand for calling LFRCDestroy() once with each argument.

(line 32) to indicate that one pointer to the SNode exists (i.e., the one returned by **new**).

Step 2 Our specific LFRCDestroy() function for Snark nodes is:

```
void LFRCDestroy(SNode *v) {
    if (v != null && add_to_rc(v, -1)==1) {
        LFRCDestroy(v\rightarrowL);
        LFRCDestroy(v\rightarrowR);
        delete v;
    }
}
```

Step 3 The Snark algorithm required a slight modification in order to ensure that there are never any cycles in garbage. The reason is that sentinel nodes have self pointers. The correctness of the Snark algorithm is not affected by changing these self pointers to null pointers, and interpreting null pointers the same way the original algorithm interprets self pointers. This is because null pointers were not used at all in the original algorithm, so every occurrence of a null pointer in the revised algorithm can be interpreted as a self pointer. The result is that there are no cycles in garbage because when a node is removed from the deque by one of the popLeft() and popRight() operations (not shown; see [3]), it becomes the new sentinel node on that side, and the outgoing pointer from this node is set to null, which breaks the cycle with the previous sentinel. To effect these changes, we changed the Snark constructor to set the dummy node up with null pointers instead of self pointers (lines 36..37); we changed the checks for self pointers to checks for null pointers (line 59); and we changed the code in the leftPop() and rightPop operations (not shown) to install null pointers instead of self pointers.

Step 4 In this case, the LFRC operations (shown in the next section) are already typed for Snark.

Step 5 We replaced all pointer accesses with LFRC operations, in accordance with the transformations shown in Table 1. (See lines 35..39, 54, 57..58, 60..62, and 65..66.)

Step 6 We inserted calls to LFRCDestroy() for each time a pointer variable is destroyed (for example, before returns from functions that contain local pointer variables). (See lines 52, 63, and 67.) We also added initialization to null for all pointers. (See lines 32, 34, and 50.) These initializations were not needed in the original algorithm, as the pointers are always modified before they are read. However, because LFRCStore() must reduce the reference count of the object previously pointed to by the pointer it modifies, it is important to ensure that pointers are initialized to null before accessing them with any LFRC operation. Finally, we added a destructor operation for Snark objects, which writes null to the LeftHat, RightHat, and Dummy pointers before destroying the Snark object itself (lines 42..44). This ensures that objects remaining in the deque are freed, which was not necessary in the GC-dependent version. (Note: this destructor is not intended to be invoked concurrently with other operations; it should be invoked only after all threads have finished accessing the deque.)

This concludes our discussion of the application of the LFRC methodology to Snark. All steps except Step 3 were quite mechanical, and did not require any creativity or reasoning about the algorithm. In this case, Step 3 was also quite easy. However, this will clearly be more difficult for some algorithms. In any case, Step 3 can be done before any of the other steps, and then all of the other steps could be automated. This would enhance our methodology from a software engineering point of view, because modifications could then be made to the original GC-dependent design and the automated steps rerun, which would prevent programmers from having to deal with the somewhat ugly code produced by the transformation described above. This would also prevent errors in this transformation, but may preclude some optimizations, such as the elision of calls to LFRCDestroy() with null pointers discussed above.

6 LFRC Implementation

In this section we describe simple implementations of the LFRC operations, and explain why they ensure that there are no memory leaks, and that memory does not get freed prematurely. The basic idea is to implement a reference count in each object that reflects the number of pointers to the object. When this count reaches zero, there are no more pointers to the object, so no more pointers to it can be created, and it can be freed.

The main difficulty is that we cannot atomically change a pointer variable from pointing to one object to pointing to another and update the reference counts of both objects. We overcome this problem by exploiting the observations that:

- provided an object's reference count is always at least the number of pointers to the object, it will never be freed prematurely, and
- provided the reference count eventually becomes zero after there are no longer any pointers to the object, the object is eventually freed.

Thus, we conservatively increment an object's reference count before creating a new pointer to it. If we subsequently fail to create that pointer, then we can decrement the reference count again afterwards to compensate. The key mechanism in our implementation is the use of DCAS to increment an object's reference count while simultaneously checking that some pointer to the object exists. This avoids the possibility of updating an object after it has been freed.

Below we describe our lock-free implementations of the LFRC operations, which are shown in Figure 2. We begin with the implementation of LFRCLoad(), which accepts two parameters: a pointer dest to a local pointer variable of the calling thread, and a pointer A to a shared pointer. This operation loads the value in the location pointed to by A into the variable pointed to by dest. This potentially has the effect of destroying a pointer to one object O1 (the one previously pointed to by *dest) and creating a pointer to another O2 (the one pointed to by *A). It is important to increment the reference count

```
void LFRCLoad(SNode **dest, SNode **A) {
                                                                           void LFRCStore(SNode **A, SNode *v) {
    long r; SNode *a, *olddest = *dest;
1
                                                                               SNode *oldval;
2
                                                                           24
     while (true) {
                                                                               if (v != null) add\_to\_rc(v, 1);
3
        a = *A;
                                                                           25
                                                                               while (true) {
        if (a == null) {
                                                                           26
                                                                                   oldval = *A;
4
5
           *dest = null:
                                                                           27
                                                                                   if (CAS(A, oldval, v)) {
           break;
                                                                           28
                                                                                      LFRCDestroy(oldval);
6
                                                                           29
                                                                                      return;
7
        r = a \rightarrow rc;
        if (r > 0 \&\& DCAS(A, \&a \rightarrow rc, a, r, a, r+1))  {
8
                                                                                }
           *dest = a;
9
10
           break:
    LFRCDestroy(olddest);
11
                                                                           void LFRCCopy(SNode **v, SNode *w) {
}
                                                                               Snode *oldv = *v;
SNode* LFRCPass(Snode *v) {
                                                                           31
                                                                               if (w != null) add_to_rc(w,1);
                                                                               v = w;
   if (v!=null) add_to_rc(v,1);
                                                                           32
13
    return v;
                                                                              LFRCDestroy(oldv);
}
void LFRCDestroy(SNode *v) {
    if (v != null && add_to_rc(v, -1)==1) {
15
        LFRCDestroy(v \rightarrow L);
                                                                           bool LFRCDCAS(SNode **A0, SNode **A1,
16
        LFRCDestroy(v \rightarrow R);
                                                                                              SNode *old0, SNode *old1,
17
                                                                                              SNode *new0, SNode *new1) {
        delete v;
                                                                           34 if (new0 != null) add_to_rc(new0, 1);
                                                                               if (\text{new1} != \text{null}) \text{ add\_to\_rc(new1, 1)};
                                                                               if (DCAS(A0, A1, old0, old1, new0, new1)) {
long add_to_rc(SNode *v, int val) {
                                                                           37
                                                                                   LFRCDestroy(old0,old1);
    long oldre;
                                                                           38
                                                                                   return true;
19
    while (true) {
                                                                                } else {
20
                                                                           39
                                                                                   LFRCDestroy(new0,new1);
        oldrc = v \rightarrow rc;
21
        if (CAS(&v→rc, oldrc, oldrc+val))
                                                                                   return false;
                                                                           40
22
           return oldrc;
     }
}
```

Fig. 2. Code for LFRC operations. LFRCCAS() is the same as LFRCDCAS(), but without the second location. LFRCStoreAlloc() is like LFRCStore() except that it does not increment the reference count of the object.

of object 02 before decrementing the reference count of 01. To see why, suppose we are using LFRCLoad(), to load the "next" pointer of a linked list node into a local variable that points to that node. We must ensure that the node is not freed (and potentially reused for another purpose) before we load the next pointer from that node. Therefore, LFRCLoad() begins by recording the previous value of *dest (line 1) for destruction later (line 11). It then loads a new value from *A, and increments the reference count of the object pointed to by *A in order to reflect the creation of a new pointer to it. Because the calling thread does not necessarily already have a pointer to this object, it is *not* safe to update the reference count using a simple CAS, because the object might be freed before the CAS executes. (Valois [25] used this approach, and as a result was forced to maintain unused nodes explicitly in a freelist, thereby preventing the space consumption of a list from shrinking over time.) Therefore, LFRCLoad() uses DCAS to attempt to atomically increment the reference count, while ensuring that the pointer to the object still exists. (A similar trick was used by Greenwald [7] in his universal constructions.) This is achieved as follows. First, LFRCLoad() reads the contents of *A (line 3). If it sees a null pointer, there is no reference count to be incremented, so LFRCLoad() simply sets *dest to null (lines 4..6). Otherwise, it reads the current reference count of the object pointed to by the pointer it read in line 3, and then attempts to increment this count using DCAS (line 8). (Note that there is no risk that the object containing the pointer being read by LFRCLoad() is freed during the execution of LFRCLoad() because the calling thread has a pointer to this object that is not destroyed during the execution of LFRCLoad(), so its reference count cannot fall to zero.) If the DCAS succeeds, then the value read is stored in *dest (line 9). Otherwise, LFRCLoad() retries.

Having successfully loaded a pointer from *A into *dest, and incremented the reference count of the object to which it points (if it is not null), it remains to call LFRCDestroy(*dest) (line 11) in order to decrement

the reference count of the object previously pointed to by *dest. As explained previously, if LFRCDestroy()'s argument is non-null, then it decrements the reference count of the object pointed to by its argument (line 14). This is done using the add_to_rc() function, which is implemented using CAS. add_to_rc() is safe (in the sense that there is no risk that it will modify a freed object) because it is called only in situations in which we know that the calling thread has a pointer to this object, which has previously been included in the reference count. Thus, there is no risk that the reference count will become zero, thereby causing the object to be freed, before the add_to_rc() function completes. If this add_to_rc() causes the reference count to become zero, then we are destroying the last pointer to this object, so it can be freed (line 17). First, however, LFRCDestroy() calls itself recursively with each pointer in the object in order to update the reference counts of objects to which the soon-to-be-freed object has pointers (lines 15 and 16).

LFRCPass() accepts one parameter, a pointer value, and simply increments the reference count of the object to which it points. The reason is that when a pointer is passed by value, this creates a new version of the pointer, which must be reflected in the reference count.

LFRCStore() accepts two parameters, a pointer A to a location that contains a pointer, and a pointer value v to be stored in this location. If the new value to be stored is not null, then LFRCStore() increments the reference count of the object to which it points (line 24). Note that at this point, the new pointer to this object has not been created, so the reference count is greater than the number of pointers to the object. However, this situation will not persist past the end of the execution of LFRCStore(), because LFRCStore() will not return until that pointer has been created. The pointer is created by repeatedly reading the current value of the pointer, and using CAS to attempt to change it to v (lines 25..29). When the CAS succeeds, we have created the pointer previously counted, and we have also destroyed a pointer, namely the previous contents of *A. Therefore, LFRCStore() calls LFRCDestroy() with the previous value of the pointer (line 28).

LFRCCopy() accepts two parameters, a pointer v to a local pointer variable, and a value w of a local pointer variable. This operation assigns the value w to the variable pointed to by v. This creates a new pointer to the object pointed to by w (if w is not null), so LFRCCopy() increments the reference count of that object (line 31). It also destroys a pointer — the previous contents of *v — so LFRCCopy() calls LFRCDestroy() with that previous value (see lines 30 and 33). LFRCCopy() also assigns the value w to the pointer variable pointed to by v (line 32).

LFRCDCAS() accepts six parameters, corresponding to the DCAS parameters described in Section 1. LFRCD-CAS is similar to LFRCStore() in that it increments the reference counts of objects before creating new pointers to them, thereby temporarily setting these counts artificially high (lines 34..35). LFRCDCAS() differs from LFRCStore(), however, in that it does not insist on eventually creating those new pointers. If the DCAS at line

36 fails, then LFRCDCAS() calls LFRCDestroy() for each of the objects whose reference counts were previously incremented, in order to compensate for those previous increments, and then returns false (lines 39..40). On the other hand, if the DCAS succeeds, then the previous increments were justified, but we have destroyed two pointers, namely the previous values of the two locations accessed by the DCAS. Therefore, LFRCDCAS() calls LFRCDestroy() in order to decrement the reference counts of these objects, and then returns true (lines 37..38). The implementation of LFRCCAS() (not shown) is just the obvious simplification of that of LFRCDCAS().

The LFRC operations are lock-free because each loop terminates unless some value changed during the loop iteration, and each time a value changes, the process that changes it terminates a loop.

In the appendix, we prove the following safety lemma:

Lemma 1: LFRC guarantees that a pointer to an object exists only if that object has been allocated and not deleted since.

The above lemma guarantees that correctly-transformed applications will never have out-of-date pointers to objects that have already been deallocated. We also prove the following progress lemma.

Lemma 2: LFRC guarantees that if an object W has not been deleted, and no usable³ pointer to W exists, then eventually W is deleted, provided no thread fails and every recursive call to LFRCDestroy() eventually returns.

This property is weaker than we might like for two reasons. The first is that it does not make any guarantees in the face of thread failures. Observe that, without special system support, we cannot guarantee perfect collection in the face of thread failures because we cannot distinguish between a thread that has failed while holding a pointer and a thread that has been delayed for some reason, but will eventually continue execution, and potentially use that pointer.

The second reason is that it does not make any guarantees in executions in which some recursive call to LFRCDestroy() does not return. This is possible because the recursion can chase a chain of objects that grows faster than LFRCDestroy() can destroy it. In this case, even though the objects on the chain will be on the stack of the thread executing LFRCDestroy(), and would therefore be deleted if the recursion unwound to the top, the algorithm as presented can lead to an unbounded number of garbage objects never being collected. This problem is not unique to LFRC, and has been addressed in various garbage collectors in the past by explicitly maintaining the set of references yet to be processed, and deleting objects before processing the references they contain [13]. Because this technique is well known, we decided not to complicate our algorithm presentation and correctness proof by incorporating it here. We present the somewhat weak progress property of Lemma 2 because it would be use-

See the appendix for the definition of a "usable" pointer.

ful in proving stronger properties about an LFRC system modified as discussed above to avoid the problems inherently associated with destroying pointers recursively.

7 Related Work

In this section we briefly discuss related work not already mentioned in the paper. In particular, we discuss previous efforts involving concurrent garbage collection, as it seems possible that techniques used in such work might be easily combined with GC-dependent data structure implementations to achieve the goal of our work. However, as discussed below, previous concurrent garbage collection techniques do not appear to be suitable for this purpose.

In [24], Steele initiated work on *concurrent* garbage collection, in which the garbage collector can execute concurrently with the mutator. Much work has been done on concurrent garbage collection since. However, in almost all such work, mutual exclusion is used between mutator threads and the garbage collector (e.g. [4]) and/or stop-the-world techniques are used. Therefore, these collectors are not lock-free.

Nonetheless, there are several pieces of prior work in which researchers have attempted to eliminate the excessive synchronization of mutual exclusion and stop-theworld techniques. The first is due to Herlihy and Moss [11]. They present a lock-free copying garbage collector, which employs techniques similar to early lock-free implementations for concurrent data structures. The basic idea in these algorithms is to copy an object to private space, update it sequentially, and later attempt to make the private copy current. This must be done each time an object is updated, a significant disadvantage. (Herlihy and Moss also outline a possible approach for overcoming this limitation, but this approach is limited to certain circumstances, and is not generally applicable.) We therefore believe that our algorithm will perform better in most cases. Nonetheless, their approach has two advantages over ours: it does not require a DCAS operation, and it can collect cyclic garbage.

Hesselink and Groote [12] designed a wait-free garbage collector. However, this collector applies only to a restricted programming model, in which a thread can modify only its roots; objects are not modified between creation and deletion. This collector is therefore not generally applicable.

Levanoni and Petrank [18] have taken another approach to attacking the excessive synchronization of most other garbage collectors. Their approach is designed to avoid synchronization as much as possible: they even eschew the use of CAS. However, their desire to avoid such synchronization has resulted in a rather complicated algorithm. We suspect that algorithms that make judicious use of strong synchronization mechanisms (such as DCAS) will perform better because the algorithms will be much simpler. However, this depends on many factors, including the implementation of the hypothetical strong primitives.

The "on-the-fly" garbage collection algorithm of Dijkstra et al. [6] (with extensions by Kung and Song [16], Ben-Ari [1], and van de SnepSchuet [26]) is an example of a concurrent collector: GC work is moved to a separate processor. A GC-dependent lock-free data structure implementation would gain many of the benefits of lock-free GC by using such a concurrent GC implementation, as long as the concurrent GC "kept up" with the need for new storage. But the overall system is not lock-free, because delaying the GC processor can delay all storage allocation requests.

As discussed in Section 1, subsequent to (and because of) our work presented in this paper, we have developed a technique for memory management for nonblocking dynamic-sized data structures that uses CAS, rather than DCAS, and is therefore applicable in most modern shared-memory multiprocessors. See [8–10] for details of this technique and its application. We have used this technique to implement a "single-location" version of LFRC, which we call SLFRC. The SLFRC methodology is identical to LFRC, except that it does not support an SLFRCDCAS() operation, for obvious reasons.

8 Concluding Remarks

Our methodology for transforming GC-dependent lock-free algorithms into GC-independent ones allows existing and future lock-free algorithms that depend on GC to be used in environments without GC, and even within GC implementations themselves. It also allows researchers to concentrate on the important features of their algorithms, rather than being distracted by often-tricky problems such as memory management; it can also provide a free solution to the ABA problem in some cases.

Our methodology is based on reference counting, and uses DCAS to update the reference count of an object atomically with a check that the object still has a pointer to it. By weakening the requirement that reference counts record the exact number of pointers to an object, we are able to separate the updates of reference counts from the updates of the pointers themselves. This allows us to support strong synchronization operations, including CAS and DCAS, on pointers.

The simplicity of our approach is largely due to the use of DCAS. This adds to the mounting evidence that stronger synchronization primitives are needed to support simple, efficient, and scalable synchronization.

The methodology presented here does have some short-comings. We believe that many of the techniques that have been used in previous work in garbage collection can be adapted to exploit DCAS or other strong synchronization mechanisms. One obvious example is to apply techniques that allow large structures to be collected incrementally. This would avoid long delays when a thread destroys the last pointer to a large structure. Another example is to integrate a tracing collector that can be invoked occasionally in order to identify and collect cyclic garbage.

LFRC has the disadvantages that it requires DCAS, which is not currently widely available, and that it can

access objects that have been deleted. It can only read them, but in some systems even this can be problematic because a page on which all objects have been deleted might be removed from the application's address space, and even read-only access to these objects would cause the application to crash. Since completing this work, we have invented a new technique for lock-free management that depends only on CAS, and prevents access of any kind to deleted objects [9,10,8]. These results were a direct result of insights gained and questions raised by the work presented in this paper.

A Correctness Proof for LFRC

In this appendix, we prove the correctness lemmas for LFRC that were presented in Section 6. We do not consider the LFRCCAS() operation in the proof; the aspects of the proof concerning the LFRCDAS() operation can easily be adapted to include the LFRCCAS() operation, but this would result in unenlightening redundancy.

The key intuition behind the algorithm and its correctness proof is that reference counts of objects are always incremented before a pointer to the object is created, and decremented after a pointer to the object is destroyed. Thus, the reference count is always at least the number of pointers to the object, and is sometimes more because some thread has incremented a reference counter in preparation for creating a pointer, but has not yet created it, or has destroyed a pointer, but has not yet decremented the reference count to reflect this destruction. Our central invariant (Invariant 3) counts the number of such threads, and relates this number to the reference count and number of "usable pointers" (defined later) to the object. We use the following notational conventions in the proof.

Notational Conventions: For an integer label s, we use p.s to denote the action with label s in thread p. Because there are no await actions in LFRC, an action labelled s of thread p is enabled if and only if p@s holds. We use p@S as shorthand for $(\exists s:s\in S::p@s)$. We use p.var to represent p's private variable var. We use C notation for pointers: specifically, *v represents the value pointed to by the pointer value v, and &W represents the value of pointers to object W. We say that object W is current in a given state t of an execution iff there is a previous event e that is an allocation of W and there is no event between event e and state e that is a deletion (i.e., deallocation) of e is noncurrent iff e is not current.

The code for LFRC is repeated in Figure 3, and has been modified in the following ways to facilitate the proof. The code for add_to_rc() has been removed, as we now consider this procedure to be atomically executed with the action that invokes it (see below). The LRCDestroy() operation has been changed to a generic one that is independent of the object type in question (we originally presented an LFRCDestroy() procedure specific to Snark nodes). Finally, we have changed our numbering system to indicate our atomicity assumptions.

Specifically, we consider each labeled action in Figure 3 to be atomic; note that each such action is consistent with the assumption of a sequentially-consistent shared-memory multiprocessor that supports load, store, CAS, and DCAS operations. (In some cases, it is necessary to take into account the rules of the LFRC methodology to see why these assumptions are justified. For example, in action p.4, a test on p.r, a DCAS, and a store to the address pointed to by the parameter p.dest are all treated as atomic. To see why this is justifiable, observe that p.r is a private variable of thread p, and also the parameter p.dest is required to be a pointer to a private variable of thread p. Thus, while in reality these actions cannot all be performed atomically, threads cannot distinguish between their atomic execution and their separate execution.)

We implicitly assume that each action of thread pupdates p's program counter in accordance with the selfexplanatory semantics of the pseudocode given for LFRC. For example, the execution of action p.13 occurs only if p@13 holds beforehand (i.e., action p.13 is enabled). If *(p.A) = p.oldval holds, then the CAS succeeds, so p.13 sets *(p.A) to p.v, and invokes LFRCDestroy() withp.oldval as an argument (i.e., it establishes $p@7 \land p.v =$ w, where w is the value of p.oldval before the action execution. On the other hand, if $*(p.A) \neq p.oldval$ holds, then the CAS fails, so the action simply establishes p@12(i.e., control returns to the top of the while loop). Similarly, application calls to the LFRC procedures establish appropriate values of p's program counter and parameters. Finally, the program counter of a thread that is executing application code is considered to be 0; initially p@0 holds for each thread p.

For convenience, we consider each call add_to_rc(p, val) as atomically adding val to $p \rightarrow rc$ and returning the previous value. It is easy to see that the add_to_rc() procedure shown in Figure 2 can be considered to have occurred atomically at the point at which the CAS via action 21 succeeds. (Note that this procedure adds val to $p \rightarrow rc$ and returns if and only if this CAS succeeds.)

To simplify our proof, we treat the object allocation system as if it manages a static set of objects that do not change type, and we consider the allocation of an object W and the following events that initialize all pointer fields of W to null and set W.rc to 1 to be atomic. In reality, unallocated objects can be coalesced and reallocated as different objects of different types and sizes, and the allocation and initialization of an object consists of several events. Specifically, the object is allocated, and a pointer to it stored in a private variable of the allocating thread, and then that thread uses that private variable to initialize various fields of the object. The LFRC methodology requires the allocating thread to complete all of these steps before invoking any other LFRC operation. Therefore, because no other thread or shared variable has a pointer to the object when it is allocated (see Invariant 3 and Claim 7), no thread can distinguish a real execution from one in which allocation and initialization is atomic. Furthermore, because none of our properties depend on the contents of a noncurrent object (either its reference count or its pointer fields), and be-

```
void LFRCLoad(SNode **dest, SNode **A) {
     long r; SNode *a, *olddest = *dest;
     while (true) {
2
        a = *A;
        if (a == null) {
            *dest = null:
            break;
3
        r = a \rightarrow rc;
        if (r > 0 \&\& DCAS(A, \&a \rightarrow rc, a, r, a, r+1))  {
            *dest = a:
            break:
     LFRCDestroy(olddest);
5
SNode* LFRCPass(Snode *v) {
    if (v!=null) add_to_rc(v,1);
    return v;
void LFRCDestroy(SNode *v) {
    if (v != null && add_to_rc(v, -1)==1) {
8
        for each pointer field f of SNode do
9
            LFRCDestroy(v \rightarrow f);
10
        delete v;
```

```
void LFRCStore(SNode **A, SNode *v) {
    SNode *oldval;
11 if (v != null) add_to_rc(v, 1);
     while (true) {
12
        oldval = *A;
13
        if (CAS(A, oldval, v)) {
           LFRCDestroy(oldval);
14
           return;
     }
void LFRCCopy(SNode **v, SNode *w) {
15 Snode *oldv = *v:
    if (w != null) add_to_rc(w,1);
     v = w;
    LFRCDestroy(oldv);
}
bool LFRCDCAS(SNode **A0, SNode **A1,
                   SNode *old0, SNode *old1,
                   SNode *new0, SNode *new1) {
    if (\text{new0} != \text{null}) \text{ add\_to\_rc(new0, 1)};
    if (\text{new1} != \text{null}) \text{ add\_to\_rc(new1, 1)};
17
18
    if (DCAS(A0, A1, old0, old1, new0, new1)) {
19
        LFRCDestroy(old0);
        LFRCDestroy(old1);
20
        return true;
     } else {
        LFRCDestroy(new0);
21
22
        LFRCDestroy(new1);
        return false;
```

Fig. 3. Code for LFRC operations modified to facilitate proof as described in text.

cause we show that the contents of a noncurrent object are never modified (see Invariant 3 and Claims 1 and 2), our assumption about the allocator maintaining a static set of fixed-type objects is unnecessary.⁴

Our model of program execution is similar to most others found in the literature, so we only sketch the important details of it here. A program's semantics is defined by a set of executions. An execution of a program is a finite or infinite sequence $t_0 \stackrel{s_0,p_0}{\longrightarrow} t_1 \stackrel{s_1,p_1}{\longrightarrow} \cdots$, where t_0 is an initial state and $t_i \stackrel{s_i,p_i}{\longrightarrow} t_{i+1}$ denotes that state t_{i+1} is reached from state t_i via the execution of action s_i by thread p_i . We call the execution of an action an event. We prove a program correct by proving that its correctness conditions hold in all executions that start from a state that satisfies the initial conditions of the program.

We consider an arbitrary execution of the application and associated calls to LFRC procedures. Because all pointer creation, destruction, modification, etc. is required to be done through LFRC procedures, we can reason about LFRC independent of the behaviour of a specific application. (However, we do assume the LFRC

methodology was correctly applied to produce the application code. One example is that the application obeys the rules about passing pointers to private pointer variables where specified. Another is that the application always destroys a pointer v after invoking LFRCDestroy(v) without using v between the return from LFRCDestroy() and the destruction of v.)

The following assumption abstracts away from and formalizes the fact that the underlying allocation mechanism behaves correctly, provided it is used correctly.

Assumption 1: If no noncurrent object has ever been modified or deleted before state s, and the allocator is invoked from state s, then the object returned by the allocator was noncurrent in state s.

As stated earlier, we argue that the reference count of an object is always at least the number of usable pointers to that object. We must first define what "usable" means, and justify this strategy. Roughly speaking, the set of usable pointers is the set of pointers that could be used by the application in the future. For convenience in the proof, we define the set of usable pointers slightly more broadly, as follows:

⁴ These justifications should technically be made at every step of the induction; we decided not to clutter the proof with this technicality.

Definition 1: alive(V, f) holds for a current object V and some pointer field f of V iff there does not exist a thread p that has invoked LFRCDestroy(V.f) via action p.9 but has not subsequently deleted V via action p.10.

Definition 2: A usable pointer to an object W is a pointer to W stored in a statically-allocated shared pointer variable, in an application private pointer variable or parameter that has not been passed to LFRCDestroy(), or in a pointer field f of a current object V such that alive(V, f) holds.

Observe that local variables and parameters of LFRC procedures (which are never visible to the application), and application private pointer variables and parameters whose values have been passed to LFRCDestroy() are not usable. (Recall that the LFRC methodology requires LFRCDestroy() to be called only with pointers that are about to be destroyed, for example because they are about to go out of scope, so the application does not use them again before their destruction.) At the moment when an application thread invokes LFRCDestroy(v), vtransitions from being a usable pointer to being an unusable pointer, so we consider this the destruction of a usable pointer. Similarly, when LFRCDestroy() invokes itself recursively with a field containing a pointer, we consider that pointer to become unusable at that moment. Finally, the return statement of the LFRCPass() procedure creates a usable pointer; this pointer is destroyed when the called procedure invokes LFRCDestroy() in preparation for its parameter going out of scope when it returns, or when LFRCLoad() or LFRCCopy() overwrites the pointer because the procedure has passed the address of the parameter to LFRCLoad() or LFRCCopy().

We prove all of our properties together by induction on the length of the execution. Therefore in the proof of each property, we are justified in assuming that all properties hold before each action is executed. For brevity, we sometimes say "by Invariant 3" when we really mean "by the inductive hypothesis that Invariant 3 holds in the state before the event we are considering". It is important to note that, in order to avoid circular reasoning, we never assume that any property holds in the state after the event we are reasoning about. (Actually, in the proof of Lemma 2, we do use the fact that some properties hold after an event. However, this lemma is not used in the proof of any other property, so there is no circular reasoning.) We now present our proof, beginning with supporting properties, and leading up to the proofs of the required lemmas.

Claim 1: If no usable pointer to an object V exists, then fields of an object V other than V.rc are not modified, and V.rc is not incremented.

Proof: By the rules about not using a pointer after invoking LFRCDestroy() with it, application code cannot modify an object unless it holds usable a pointer to that object. Thus, it remains to consider actions of LFRC. We consider events that modify $V \rightarrow f$ for some field f of V other than v.rc, and events that modify V.rc.

The only events that modify V.f are the execution of action p.13 while p.A = &V.f and the execution of action p.18 while $p.A0 = \&V.f \lor p.A1 = \&V.f$. In each case, the address passed to the corresponding LFRC procedure (LFRCStore() or LFRCDCAS()) was acquired by the calling procedure from a private variable or parameter that points to V; the application code still has this pointer when the events in question occur.

Only the execution of actions p.4, p.6, p.11, p.15, p.16, and p.17 for some p can increment V.rc. For each of these events except p.4, the pointer passed to $add_to_rc()$ is a pointer value that was passed by application code to the relevant LFRC procedure. Because of the restrictions on how applications manipulate and store pointers, these pointers were passed from private variables or parameters of the application thread invoking the LFRC procedure. Thus, these pointers still exist at the time of the execution of the event that increments W.rc. Furthermore, none of these events is in LFRCDestroy(), so these pointers are usable.

Action p.4 increments V.rc only if executed when $*(p.A) = p.a \land p.r > 0 \land p.a = \&V$ holds. The parameter p.A is either a pointer to a statically-allocated shared pointer variable or a pointer to a pointer field of an object to which the invoking thread has a usable pointer. As before, the usable pointer in the latter case still exists. Thus, in both cases, some usable pointer to V exists.

Claim 2: V.rc is decremented only if $p@7 \land p.v = \&V$ holds for some thread p.

Invariant 1: $p@\{2..4\} \Rightarrow p.olddest = *(p.dest)$

Proof: Initially p@0 holds, so the invariant holds. Only the execution of action p.1 establishes the antecedent, and it also establishes the consequent. No action modifies p.olddest or *(p.dest) while the antecedent holds (recall that the first argument to LFRCLoad() is the address of a private pointer variable, and therefore the contents of the variable cannot be changed by events of other threads).

Definition 3: We use destroying(p, s, v, val) as shorthand for "thread p previously invoked LFRCDestroy() via action p.s when p.v = val held, and has not yet returned from that call".

The following definition is used to capture the value of W.rc for an object W, as shown in Invariant 3 below.

Definition 4: $SUM_W \equiv$

```
\# of usable pointers to W +
                                                                                                                                                                                                                                                                                                                                                                                                                    (A)
 |\{p: p@\{12..13\} \land p.v = \&W\}| +
                                                                                                                                                                                                                                                                                                                                                                                                                    (B)
  |\{p: p@\{17..18\} \land p.new0 = \&W\}| +
                                                                                                                                                                                                                                                                                                                                                                                                                      (C)
 |\{p: p@18 \land p.new1 = \&W\}| +
                                                                                                                                                                                                                                                                                                                                                                                                                    (D)
  |\{p: p@7 \land p.v = \&W\}| + |\{p: p@7 \land p.v = \&V\}| + |\{p: p~7 \land p.v = \&V\}| + |\{p:
                                                                                                                                                                                                                                                                                                                                                                                                                    (E)
  |\{p: p@5 \land p.olddest = \&W\}| +
                                                                                                                                                                                                                                                                                                                                                                                                                    (F)
    \{p: p@19 \land p.old0 = \&W\} | +
                                                                                                                                                                                                                                                                                                                                                                                                                    (G)
     \{p: p@19 \land p.old1 = \&W\} | +
                                                                                                                                                                                                                                                                                                                                                                                                                    (H)
  |\{p: p@21 \land p.new0 = \&W\}| +
                                                                                                                                                                                                                                                                                                                                                                                                                    (I)
                                                                                                                                                                                                                                                                                                                                                                                                                 (J)
|\{p: p@21 \land p.new1 = \&W\}| +
```

$$|\{p: destroying(p, 19, old1, \&W)\}| + \tag{K}$$

$$|\{p: destroying(p, 21, new1, \&W)\}| +$$
 (L)

$$\{p: p@20 \land p.old1 = \&W\} | +$$
 (M)

$$|\{p: p@22 \land p.new1 = \&W\}|$$
 (N)

Before proving our main invariant about SUM_W , we first need some supporting properties.

Claim 3: Every event that increments W.rc also increases SUM_W by one.

Proof: We argue the claim for the enabled event of some thread p for each of the following situations (it is easy to see that this covers all events that increment W.rc):

- $p@4 \land p.r > 0 \land *(p.A) = p.a \land p.a \rightarrow rc = p.r \land p.a = \&W$
 - Let n be 1 if *(p.dest) = &W and 0 otherwise. The net effect on (A), the number of usable pointers to W, is to increase it by 1-n. By Invariant 1, if n=1, then this event increases (F) by 1, and if n=0, (F) is not affected. In either case, quantities of SUM_W other than (F) and the number of usable pointers to W, are unaffected by this event. Thus, this event increases SUM_W by 1.
- $p@6 \wedge p.v = \&W$. This event increases (A), the number of usable pointers to W, by 1 and does not affect any other quantity in SUM_W .
- $p@11 \land p.v = \&W$. This event increases (B) by 1 and does not affect any other quantity in SUM_W .
- p@15 ∧ p.w = &W.
 Let n be 1 if *(p.v) = &W and 0 otherwise. The net effect on the number of usable pointers to W is to increase it by 1-n. If n = 1, then this event increases (E) by 1, and if n = 0, (E) is not affected. In either case, quantities of SUM_W other than (E) and the number of usable pointers to W, are unaffected by this event. Thus, this event increases SUM_W by 1.
- $p@16 \land p.new0 = \&W$. This event increases (C) by 1 and does not affect any other quantity in SUM_W .
- $p@17 \wedge p.new1 = \&W$. This event increases (D) by 1 and does not affect any other quantity in SUM_W .

Claim 4: Every event that decrements W.rc also decreases SUM_W by one.

Proof: The only event that decreases W.rc is the execution of action p.7 while p.v = &W. This event reduces (E) by 1. If W.rc = 1, then this event does not affect any other quantity in SUM_W . If $W.rc \neq 1$, then this event may decrease (K) by 1, in which case it also increases (M) by 1, or it may decreases (L) by 1, in which case it also increases (N) by 1.

Claim 5: Events other than the allocation and initialization of W and events that modify W.rc do not change the value of SUM_W .

Proof: We consider all events that create or destroy a usable pointer to W, and all events that potentially change

one of the quantities (B) through (N). By Definition 2, a usable pointer to an object W can be created only by:

- storing a pointer to W in a statically-allocated shared variable (via LFRCStore() or LFRCDCAS())
- storing a pointer to W in an application private variable or parameter (via LFRCLoad(), LFRCCopy(), passing a pointer to W to an application procedure, or allocating and initializing W)
- storing a pointer to W in a field f of a current object V such that alive(V, f) holds (via LFRCStore() or LFRCDCAS())
- establishing alive(V, f) for some current object V and some field f of V such that V.f = &W holds afterwards
- causing some noncurrent object V to become current such that, for some field f of V, $V.f = \&W \land alive(V, f)$ holds afterwards

Similarly, a usable pointer to an object W can be destroyed only by:

- storing a pointer value to a statically-allocated shared variable that contains a pointer to W (via LFRCStore() or LFRCDCAS())
- storing a pointer value to an application private variable or parameter that contains a pointer to W (via LFRCLoad(), LFRCCopy(), or allocating and initializing some object and storing the result in a private pointer variable or parameter that contains a pointer to W)
- the application invoking LFRCDestroy() passing a pointer to W in preparation for a private variable or parameter that contains it being destroyed (for example by going out of scope)
- storing a pointer value to a field f of a current object V such that $V.f = \&W \land alive(V, f)$ holds (via LFRCStore() or LFRCDCAS())
- falsifying alive(V, f) for some current object V and some field f of V such that V.f = &W holds
- causing some current object V such that alive(V, f)
 holds for some field f of V and V.f = &W holds to
 become noncurrent

Observe that some events can simultaneously create and/or destroy usable pointers to W by storing a new pointer value in a location that already contains a pointer. In cases where a pointer to W is stored into a location that already contains a usable pointer to W, we consider that one usable pointer to W has been created and another destroyed, with no net effect on the number of usable pointers to W.

We first address events that establish or falsify alive(V, f) for some current object V and field f of V, and events that cause some object V to become current or noncurrent. alive(V, f) can be established only by some thread p deleting V via action p.10. However, this also causes V to become noncurrent, and therefore does not create any usable pointers to W. alive(V, f) can be falsified only by some thread p invoking LFRCDestroy(V.f); all such events are considered below. An object V becomes current only as a result of its allocation and initialization;

the claim does not require us to prove anything about this case. An object V becomes noncurrent only as a result of some thread p deleting it via action p.10; if this action is enabled, then alive(V, f) does not hold for any field f of V, so this event does not destroy any usable pointer to W.

It remains to consider the following events:

- The allocation and initialization of W, and subsequent storing of the resulting pointer to W in a private pointer variable or parameter.
 - The claim does not require us to prove anything about this event.
- An application thread invokes LFRCDestroy(), passing a pointer to W as an argument.
 This event destroys one usable pointer to W, and therefore decreases (A) by one, but balances this by increasing (E) by one, and does not affect any other quantity of SUM_W.
- $p@2 \wedge *(p.A) = \text{null} \wedge *(p.\text{dest}) = \&W$. This event destroys one usable pointer to W and, by Invariant 1, increases (F) by one.
- $p@4 \land p.r > 0 \land *(p.A) = p.a \land p.a \rightarrow rc = p.r$ If p.a = &W, then this event modifies W.rc, so we have no proof obligation. Otherwise, $p.a \neq \&W$. If *(p.dest) = &W, then this event destroys one usable pointer to W and, by Invariant 1, increases (F) by one. If $*(p.dest) \neq \&W$, then this event does not affect the number of usable pointers to W, and does not affect F. In any case, this event does not affect quantities of SUM_W other than (F) and the number of usable pointers to W.
- $p@6 \wedge p.v = \&W$ This event modifies W.rc, so we have no proof obligation.
- p@9 ∧ (p.v)→f = &W
 This event destroys a usable pointer to W (see introductory discussion of usable pointers), increases
 (E) by 1, and does not affect any other quantity of SUM_W.
- $p@13 \wedge *(p.A) = p.oldval$. Let n be 1 if *(p.A) = &W and 0 otherwise. Let m be 1 if *(p.v) = &W and 0 otherwise. The net effect on the number of usable pointers to W is to increase it by m-n. If n=1, then this event increases (E) by 1, and if n=0, then (E) is unaffected. Also, if m=1, then this event decreases (B) by 1, and if m=0, then (B) is unaffected. In any case, quantities of SUM_W other then (B), (E), and the number of usable pointers to W are unaffected.
- If p.w = &W, then this event modifies W.rc, so we have no proof obligation. Otherwise, $p.w \neq W$. If *(p.v) = &W then this event reduces the number of usable pointers to W by 1, and increases (E) by 1. Otherwise, it does not affect the number of usable pointers to W, and does not affect (E). In any case, quantities of SUM_W other than (E) and the number of usable pointers to W are unaffected.
- $p@18 \land (*(p.A0) \neq p.old0 \lor *(p.A1) \neq p.old1)$

Let n be 1 if p.new0 = &W and 0 otherwise. Let m be 1 if p.new1 = &W and 0 otherwise. This event reduces (C) by n and increases (I) by n, and reduces (D) by m and increases (J) by m. This event does not create or destroy any usable pointers to W, and does not affect quantities of SUM_W other than (C), (D), (I), and (J).

• $p@18 \wedge *(p.A0) = p.old0 \wedge *(p.A1) = p.old1$ Let n_0 be 1 if *(p.A0) = &W and 0 otherwise. Let n_1 be 1 if *(p.A1) = &W and 0 otherwise. Let m_0 be 1 if *(p.new0) = &W and 0 otherwise. Let m_1 be 1 if *(p.new1) = &W and 0 otherwise. The net effect on the number of usable pointers to W is to increase it by $m_0 + m_1 - (n_0 + n_1)$. If n_0 is 1, then this event increases (G) by 1, and if $n_0 = 0$, then (G) is unaffected. If n_1 is 1, then this event increases (H) by 1, and if $n_1 = 0$, then (H) is unaffected. If m_0 is 1, then this event decreases (C) by 1, and if $m_0 = 0$, then (C) is unaffected. If m_1 is 1, then this event decreases (D) by 1, and if $m_1 = 0$, then (D) is unaffected. In any case, quantities of SUM_W other than (C), (D), (G), (H), and the number of usable pointers to W are unaffected.

Having considered the effect on SUM_W of all events that create or destroy usable pointers to W, we must also consider additional events that potentially affect quantities (B) through (N). We begin with events that increase quantities (B) through (N).

We first consider quantities (K) and (L). The only event that increases (K) is the execution of action p.19 while p.old1 = &W holds. This event also decreases (H) by 1. Also, if p.old0 = &W holds, then this event decreases (G) by 1 and increases (E) by 1; otherwise, it does not affect either quantity. In any case, this event does not modify quantities other than (K), (H), (G), and (E).

The only event that increases (L) is the execution of event p.21 while p.new1 = &W holds. This event also decreases (J) by 1. Also, if p.new0 = &W holds, then this event decreases (I) by 1 and increases (E) by 1; otherwise, it does not affect either quantity. In any case, this event does not modify quantities other than (L), (J), (I), and (E).

We now consider quantities (B) through (J) and (M) through (N). None of the local variables in any of these quantities is modified while the constraints on p's program counter corresponding to that quantity is true, so we need only consider events that establish the constraints on the program counter.

Only action p.11 for some p increases (B), and it does so only if executed when p.v = &W, in which case it modifies W.rc.

Only action p.16 for some p increases (C), and it does so only if executed when p.new0 = &W, in which case it modifies W.rc.

Only action p.17 for some p increases (D), and it does so only if executed when p.new1 = &W, in which case it modifies W.rc.

Only an invocation of LFRCDestroy() with W by the application, and the execution of the enabled event of thread p in one of the following cases increases (E):

- $p@9 \land (p.v) \rightarrow f = \&W$
- $p@13 \wedge *(p.A) = p.oldval \wedge p.oldval = \&W$
- $p@15 \wedge *(p.v) = \&W$
- $p@5 \wedge p.olddest = \&W$
- $p@19 \land p.old0 = \&W$
- $p@20 \land p.old1 = \&W$
- $p@21 \land p.new0 = \&W$
- $p@22 \land p.new1 = \&W$

We have already shown above that the application invoking LFRCDestroy() with a pointer to W does not change the value of SUM_W , and neither do any of the first three events listed above. Each of the remaining five events increases (E) by 1, and these events respectively reduce the following quantities by 1 and do not affect any other quantity of SUM_W : (F), (G), (M), (I), and (N).

Only actions p.2 and p.4 for some p increase (F). We have already shown above that p.2 does not change the value of SUM_W if executed when $p@2 \wedge *(p.A) =$ null $\wedge *(p.dest) = \&W$ holds. If this event is executed when $p@2 \wedge (*(p.A) \neq \text{null} \vee *(p.dest) \neq \&W)$ holds, then it does not affect any quantity of SUM_W .

We have already shown above that p.4 does not change the value of SUM_W if executed when $p@4 \land p.r > 0 \land *(p.A) = p.a \land p.a \rightarrow rc = p.r$ holds. If this event is executed when $p@4 \land (p.r \leq 0 \lor *(p.A) \neq p.a \lor p.a \rightarrow rc = p.r)$ holds, then it does not affect any quantity of SUM_W .

Only action p.18 for some p increases (G), (H), (I), or (J); we have already shown above (in two cases) that this event does not change the value of SUM_W .

Only actions that return from a call by some p to LFRCDestroy() in action 19 when p.old1 = &W held increases (M). This event also decreases (K) by 1, and does not affect any other quantity of SUM_W .

Only actions that return from a call by some p to LFRCDestroy() in action 21 when p.new1 = &W held increases (N). This event also decreases (L) by 1, and does not affect any other quantity of SUM_W .

(Note the previous two cases assume that values on the stack below the top of the stack when the call was made do not change before the return. This is justified because calls to LFRCLoad() and LFRCCopy() can only pass pointers to local variables and parameters and global private variables, as discussed in Section 3.)

It remains to consider additional events that potentially decrease quantities (B) through (N).

For each of the quantities (B) through (N) it is straightforward to identify the events that potentially decrease that quantity.

The events that decrease quantities (B) through (D) have all been considered above. (Note that if $p@13 \land *(p.A) \neq p.oldval$, then the CAS in action p.13 fails, and this event does not decrease (B) in this case because it does not falsify $p@\{12..13\}$.)

The only event that decreases (E) modifies W.rc.

Each event that decreases any of quantities (F), (G), (I), (M), and (N) by 1 also increases (E) by 1 and does not affect any other quantity in SUM_W .

The only event that decreases (H) by 1 also increases (K) by 1 and either decreases (G) by 1 and increases (E) by 1, or leaves both quantities unchanged. In either case, it does not affect any other quantity in SUM_W .

The only event that decreases (J) by 1 also increases (L) by 1 and either decreases (I) by 1 and increases (E) by 1, or leaves both quantities unchanged. In either case, it does not affect any other quantity in SUM_W .

Events that decrease (K) by 1 also increase (M) by 1, and do not affect any other quantities of SUM_W . Similarly, events that decrease (L) by 1 also increase (N) by 1, and do not affect any other quantities of SUM_W . \square

Below we prove some simple properties that show that the memory allocation system is used correctly (for example, LFRC does not allow modifications to unallocated objects, and does not delete unallocated objects). This is needed to ensure that the memory allocation system behaves correctly: only unallocated objects are returned by the allocator.

Claim 6: If no usable pointer to an object W exists, then no event other than the allocation and initialization of W increases SUM_W .

Proof: By Claims 3, 4, and 5, we need only consider events that increment W.rc. By Claim 1, no such event occurs while no usable pointer to W exists.

Definition 5: We say thread p is about to delete W iff p has decremented W.rc to 0 via action p.7 and has not yet deleted W via action p.10.

Invariant 2: If some thread p is about to delete W, then W is current, and W.rc = 0, and no thread $q \neq p$ is about to delete W.

Proof: Initially, there is no thread p that is about to delete W. The antecedent is established only by some event causing thread p to be about to delete W when no thread is about to delete W. In this case, no thread $q \neq p$ is about to delete W after this event. Also, by definition of the event, W.rc = 0 holds after it. The event can only be the execution of statement p.7 when $p@7 \land p.v = \&W$ holds. This implies that $SUM_W > 0$, which by Invariant 3, implies that W is current. Action p.7 does not make W noncurrent.

It remains to consider events that make W noncurrent, events that falsify W.rc = 0, and events that cause some thread $q \neq p$ to be about to delete W while p is about to delete W, W is current, W.rc = 0, and no thread $q' \neq p$ is about to delete W.

W is caused to become noncurrent only by a thread that is about to delete W deleting W. Because no thread other than p is about to delete W, only p can cause W to become noncurrent, which also falsifies the antecedent (i.e., p is no longer about to delete W after p deletes W).

By Invariant 3, if W is current and W.rc = 0 holds, then no usable pointer to W exists and no thread p exists such that $p@7 \wedge p.v = \&W$ holds. Therefore, by Claims 1 and 2, no event falsifies W.rc = 0 while W is current.

No thread can become about to delete W while W.rc=0. \square

Claim 7: Objects returned by the allocator are always noncurrent in the state from which the allocator is invoked.

Proof: By Invariant 2, only current objects are deleted. By Invariant 3, if W is noncurrent, then there exists no usable pointer to W and no thread p such that $p@7 \land p.v = \&W$. Thus, by Claims 1 and 2, W is never modified while it is noncurrent. Therefore, the claim follows from Assumption 1.

Claim 8: If no usable pointer to W exists, then no event other than the allocation and initialization of W creates a usable pointer to W.

Proof: We show for each event other than the allocation and initialization of W that can create a usable pointer to W that it occurs only when a usable pointer to Walready exists. The events to consider are the execution of actions p.4, p.6, p.13, p.15, and p.18 for some p. For each of these events except the first, each usable pointer created is a copy of a pointer value that was passed by application code to the relevant LFRC procedure. Because of the restrictions on how applications manipulate and store pointers, these pointers were passed from private variables or parameters of the application thread invoking the LFRC procedure. Thus, these pointers still exist at the time of the execution of the event that creates the new usable pointer. Because these pointers came from the application, and because none of these events is in LFRCDestroy(), they are usable pointers.

It remains to consider the execution of action p.4. This event creates a usable pointer to W only if $*(p.A) = p.a \land p.a = \&W$. The parameter p.A is either a pointer to a statically-allocated shared pointer variable or a pointer to a pointer field of an object to which the invoking thread has a usable pointer. As before, the usable pointer in the latter case still exists. Thus, in both cases, some usable pointer to W exists.

Invariant 3: If W is noncurrent, then $SUM_W = 0$ and if W is current, then $W.rc = SUM_W$.

Proof: Initially, W is noncurrent and no usable pointers to W exist and for all p, p@0 holds, so the invariant holds.

Only the deletion of W causes W to become noncurrent. By Invariant 2 and the inductive hypothesis, $SUM_W = 0$ in this case; by Claim 5, deleting W does not change the value of SUM_W . By Claim 6, no event falsifies $SUM_W = 0$ without causing W to become current (note that $SUM_W \ge 0$ holds by definition).

Only the allocation and initialization of W (recall that we consider this to be atomic) causes W to become current. Also, by Claim 7 and the inductive hypothesis, $SUM_W = 0$ holds before this event, and this event establishes W.rc = 1, creates one usable pointer to W,

and does not affect quantities (B) through (N). Thus, $W.rc = SUM_W = 1$ holds afterwards, as required.

By Claims 3, 4, and 5, no event falsifies $W.rc = SUM_W$ while W is current. \square

Lemma 1: LFRC guarantees that if an object W is noncurrent, then no usable pointer to W exists.

Proof: Initially, no usable pointer to W exists. Only the deletion of W causes it to become noncurrent. By Invariants 2 and 3, no usable pointer to W exists before W is deleted; the deletion of W does not create any usable pointers to W. By Claim 8, no usable pointer to W is created while W is noncurrent.

Lemma 2: LFRC guarantees that if an object W is current, and no usable pointer to W exists, then eventually W is deleted, provided no thread fails and every recursive call to LFRCDestroy() eventually returns.

Proof: It is easy to see that, if some thread p is about to delete W (see Definition 5), then p eventually deletes W provided p does not fail, and each of p's recursive calls to LFRCDestroy() eventually returns. Therefore, it suffices to argue that some thread is eventually about to delete W, which by Definition 5 occurs iff some thread decrements W.rc to zero.

Recall that when W became current, W.rc became 1. Therefore, if $W.rc \leq 0$ then some thread has already decremented W.rc to 0 (observe that W.rc is only ever decremented by 1). Therefore, consider a state in which W is current, no usable pointers to W exist, and $W.rc \geq 1$. By Invariant 2, no thread is about to delete W in this state. Also, no thread will be about to delete W until some thread decrements W.rc to zero, in which case the claim holds as argued above. Therefore, by Claims 7 and 8, if the claim does not hold, then no usable pointer to W ever exists again. By Invariant 3, we therefore need only argue that eventually each of quantities (B) through (N) of SUM_W will be zero. To show this, we group these quantities into four groups as follows:

Group 1 (B), (C), (D), (F), (G), (H), (I), and (J) Group 2 (K) and (L) Group 3 (M) and (N) Group 4 (E)

Below we argue that each quantity eventually decreases, and that a quantity can increase only if another quantity in a lower-numbered group decreases by the same amount. This implies that eventually each quantity in Group 1 is zero, and that at some point at or after that time, each quantity in Group 2 is zero, and so on. Thus, eventually, all quantities are zero, so SUM_W becomes zero.

It is easy to see that each non-zero quantity (B) through (N) eventually decreases, provided no thread fails and every recursive call to LFRCDestroy() eventually returns.

Each quantity in Group 1 increases only if a usable pointer to W exists. To see why, observe the following. First, quantities (B) through (D) increase only if an LFRC parameter that is a copy of an application

private pointer variable that points to W exists. By Invariant 1, (F) similarly increases only if the application has a usable pointer to W. Quantity (G) increases only if the DCAS in action p.18 succeeds for some thread p while $p.old\theta = \&W$ holds, which can occur only if $*(p.A0) = p.old\theta$ holds. In this case, because p.A0 is a pointer to a statically-allocated application pointer variable or to a field of an object to which p has a usable pointer, a usable pointer to W exists. Similarly, (H) increases only if a usable pointer to W exists. Quantity (I) increases only if the DCAS in action p.18 fails for some p while $p.new\theta = \&W$, in which case p still has a usable pointer to W because it passed this pointer from a private pointer variable to LFRCDCAS(). Similarly, (J) increases only if some thread has a usable pointer to W.

Quantity (K) increases only if quantity (H) decreases, and quantity (L) increases only if quantity (J) decreases (observe that (H) and (J) are in lower-numbered groups than (K) and (L)).

Quantity (M) increases only if quantity (K) decreases, and quantity (N) increases only if quantity (L) decreases (observe that (K) and (L) are in lower-numbered groups than (M) and (N)).

Quantity (E) increases only as a result of events that are invocations of LFRCDestroy() with a parameter that is a pointer to W. In the case of application invocations of LFRCDestroy(), and executions of statements p.13and p.15 for some p, usable pointers to W exist, as argued previously. In the case of a recursive invocation of LFRCDestroy() by statement p.9 with $p.v \rightarrow p.f = \&W$, p is about to delete some object V at the moment that it invokes LFRCDestroy() recursively. By Invariant 2, V is current and no other thread is about to delete V at that moment. Therefore, by Defintions 1 and 2, a usable pointer to W exists at that moment. For all other events that are invocations of LFRCDestroy() with a parameter equal to &W, one of quantities (F), (G), (I), (M), and (N) decreases, all of which are in lower-numbered groups than (E).

David L. Detlefs is a Senior Staff Engineer and the Principal Investigator of the Java Thechnology Research Group in Sun Labs. He obtained an S.B. degree from MIT, then worked briefly for TRW and as a staff programmer at the MIT Laboratory for Computer Science. Five years of effort yielded a Ph.D. from Carnegie Mellon in 1990. Then he joined the Digital Equipment Corporation's Systems Research Center, working on garbage collection and program verification until joining Sun in 1996. At Sun he has worked on various aspects of Java Virtual Machine implementation, including garbage collection, JIT compilation, and synchronization.

Paul A. Martin joined Sun Microsystems Laboratories in 1993 as founding PI and later co-PI of the Speech Applications project. He is currently a member of the Knowledge Technology Group, adapting and extending natural language and knowledge representation tools. Additionally, Paul works with the Scalable Synchronization Research Group, exploring nonblocking interaction for concurrent processes. Before join-

ing Sun, Paul pursued his research interest in using human language to communicate with computers in IBM's Austin Information Retrieval Tools group, in the Human Interface and Natural Language groups at MCC, and at SRI's Artificial Intelligence Center. He received his Ph.D. from Stanford University for research conducted at Stanford's AI Lab and at Xerox PARC, and his B.S. from North Carolina State in CS and EE.

Mark Moir received the B.Sc. (Hons.) degree in Computer Science from Victoria University of Wellington, New Zealand in 1988, and the Ph.D. degree in Computer Science from the University of North Carolina at Chapel Hill, USA in 1996. From August 1996 until June 2000, he was an assistant professor in the Department of Computer Science at the University of Pittsburgh. In June 2000, he joined Sun Microsystems Laboratories, where he is now the Principal Investigator of the Scalable Synchronization Research Group.

Dr. Moir's main research interests concern practical and theoretical aspects of concurrent, distributed, and real-time computing. His current research focuses on mechanisms for non-blocking synchronization in shared-memory multiprocessors.

Guy L. Steele, Jr. is a Distinguished Engineer at Sun Microsystems Laboratories in Burlington, Massachusetts. He is responsible for research in programming languages, parallel algorithms, implementation strategies, and architectural and software support. He has worked with James Gosling and Bill Joy on the detailed specification of the Java programming language. He has published more than two dozen papers on the subject of the Lisp language and Lisp implementation, including a series with Gerald Jav Sussman that defined the Scheme dialect of Lisp. He is an ACM Fellow and a Fellow of the AAAI; he was awarded the ACM Grace Murray Hopper Award in 1988 and a Gordon Bell Prize in 1990. He designed the original EMACS command set and was the first person to port TeX. Prior to joining Sun, he was a senior scientist at Thinking Machines Corporation, a member of technical staff at Tartan Laboratories, and an assistant professor at Carnegie-Mellon University. He is a co-author of three books on programming languages: "Common Lisp: The Language," "C: A Reference Manual," and "The High Performance Fortran Handbook." In March 2001, Guy was elected to the National Academy of Engineering.

References

- M. Ben-Ari. On-the-fly garbage collection: New algorithms inspired by program proofs. In M. Nielsen and E. Meineche Schmidt, editors, Automata, Languages and Programming, 9th Colloquium, volume 140 of Lecture Notes in Computer Science, pages 14–22, Aarhus, Denmark, 12–16 July 1982. Springer-Verlag.
- 2. D. Detlefs. Garbage collection and run-time typing as a C++ library. In *Proceedings of the 1992 Usenix C++ Conference*, pages 37–56, 1992.
- D. Detlefs, C. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. Steele. Even better DCAS-based concurrent deques. In *Proceedings of the 14th Interna-*

- tional Conference on Distributed Computing, pages 59–73, 2000.
- J. DeTreville. Experiences with concurrent garbage collectors for modula-2+. Technical Report 64, Digital Equipment Corporation Systems Research Center, 1990.
- Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In ACM SIGPLAN international symposium on Memory management. ACM Press, 2002.
- E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *CACM*, 21(11):966–975, November 1978.
- M. Greenwald. Non-Blocking Synchronization and System Design. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, August 1999.
- M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Dynamic-sized lock-free data structures. Technical Report TR-2002-110, Sun Microsystems Laboratories, 2002.
- M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. Technical Report TR-2002-112, Sun Microsystems Laboratories, 2002.
- M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. In *Proceedings of the* 16th International Symposium on DIStributed Computing, 2002. To appear.
- M. Herlihy and E. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3), 1992.
- W.H. Hesselink and J.F. Groote. Waitfree distributed memory management by create, and read until deletion (CRUD). In 113, page 17. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, December 31 1998. SEN (Software Engineering (SEN)).
- Richard E. Jones. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.
- D. E. Knuth. The Art of Computer Programming: Fundamental Algorithms. Addison-Wesley, 1968.
- H. Kung and L. Lehman. Concurrent manipulation of binary search trees. ACM Transactions on Database Systems, 5(3):354–382, 1980.
- H. T. Kung and S. Song. An efficient parallel garbage collector and its correctness proof. Technical report, Carnegie Mellon University, September 1977.
- 17. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.
- Y. Levanoni and E. Petrank. A scalable reference counting garbage collector. Technical Report Technical Report CS-0967, Computer Science Department, Technion, Israel, 1999.
- M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing, pages 267–276, 1996.
- 20. M. Moir. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.

- Motorola. MC68020 32-bit microprocessor user's manual. Prentice-Hall, 1986.
- W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Department of Computer Science, University of Maryland, 1989.
- N. Shavit and D. Touitou. Software transactional memory. Distributed Computing, Special Issue(10):99–116, 1997.
- G.L. Steele. Multiprocessing compactifying garbage collection. Communications of the ACM, 18(9):495–508, 1975.
- 25. J. Valois. Lock-free linked lists using compare-and-swap. In Proceedings of the 14th Annual ACM Symposium on Principles of Dsitributed Computing, pages 214-22, 1995. See http://www.cs.sunysb.edu/~valois for errata.
- J. L. A. van de Snepscheut. Algorithms for on-the-fly garbage collection revisited. *Information Processing Let*ters, 24(4):211–216, March 1987.