

DCAS-Based Concurrent Deques

Ole Agesen*¹ David L. Detlefs[†] Christine H. Flood[†] Alexander T. Garthwaite[†]
Paul A. Martin[†] Mark Moir[†] Nir N. Shavit^{‡1} Guy L. Steele Jr.[†]

**VMware*

[‡]*Tel Aviv University*

[†]*Sun Microsystems Laboratories*

Abstract

The computer industry is currently examining the use of strong synchronization operations such as double compare-and-swap (DCAS) as a means of supporting non-blocking synchronization on future multiprocessor machines. However, before such a strong primitive will be incorporated into hardware design, its utility needs to be proven by developing a body of effective non-blocking data structures using DCAS.

As part of this effort, we present two new linearizable non-blocking implementations of concurrent deques using the DCAS operation. The first uses an array representation, and improves on previous algorithms by allowing uninterrupted concurrent access to both ends of the deque while correctly handling the difficult boundary cases when the deque is empty or full. The second uses a linked-list representation, and is the first non-blocking unbounded-memory deque implementation. It too allows uninterrupted concurrent access to both ends of the deque. We have proved these algorithms correct with the aid of a mechanical theorem prover; we describe these proofs in the paper.

1 Introduction

In academic circles and in industry, it is becoming evident that non-blocking algorithms can deliver significant performance [4, 29, 26] and resilience benefits [16] to parallel systems. Unfortunately, there is a growing realization that existing synchronization operations on single memory locations, such as compare-and-swap (CAS), are not expressive enough to support design of efficient non-blocking algorithms [16, 17, 21], and software emulations of stronger primitives from weaker ones are still too complex to be considered practical [1, 5, 8, 9, 30]. In response, industry is currently examining the idea of supporting stronger synchronization operations in hardware. A leading candidate among such operations is double compare-and-swap (DCAS), a CAS performed atomically on two memory locations. However, before such a primitive can be incorporated into processor de-

sign, it is necessary to understand how much of an improvement it actually offers. One step in doing so is developing a body of efficient data structures based on the DCAS operation.

This paper presents two novel designs of non-blocking linearizable concurrent double-ended queues (*deques*) using DCAS. The first represents the deque in an array, while the second uses a linked list representation. The latter approach has the obvious advantage of avoiding fixed, static resource allocations.

Dequeues, originally described in [23], and currently used in load balancing algorithms [4], are classic structures to examine, in that they involve all the intricacies of LIFO stacks and FIFO queues, with the added complexity of handling operations originating at both ends of the deque. By being linearizable [22] and non-blocking [19], our concurrent deque implementations are guaranteed to behave as if operations on them were executed in a mutually exclusive manner, without actually using any form of mutual exclusion.

1.1 Related Work

Massalin and Pu [25] were the first to present a collection of DCAS-based concurrent algorithms. They built a lock-free operating system kernel based on the DCAS operation (CAS2) offered by the Motorola 68040 processor, implementing structures such as stacks, queues, and linked lists.

Greenwald, a strong advocate for using DCAS, built a collection of DCAS-based concurrent data structures improving on those of Massalin and Pu. He proposed various implementations of the DCAS operation in software and hardware, and presented two array-based concurrent deque algorithms using DCAS [16]. Unfortunately, his algorithms use DCAS in a restrictive way. The first (pages 196-197 of [16]) uses the two-word DCAS as if it were a three-word operation, keeping the two deque end pointers in the same memory word, and DCAS-ing on it and a second word containing a value. Apart from the fact that this limits applicability by cutting the index range to half a memory word, it also prevents concurrent access to the two deque ends. In the second algorithm (pages 219-220 of [16]), the push operation for one end of the deque does not check for boundary conditions, and therefore this algorithm requires an unbounded array. As far as we can tell, it would be straightforward to modify the algorithm to correctly handle overflow at both ends. A more important shortcoming of this algorithm is that it can fail to push a new value onto one of the ends, even when the deque contains only a single element, regardless of the array size.

¹Work done while a member of Sun Microsystems Laboratories.

Arora et al. [4] present an elegant CAS-based deque with applications in job-stealing algorithms. In this application, one side of the deque is accessed by only a single processor, and the other side allows only pop operations. Arora et al. exploited these restrictions to create a non-blocking implementation that requires only CAS operations for synchronization.

To the best of our knowledge, ours is the first linked-list-based lock-free deque implementation that supports concurrent access to the two ends of the deque. Since the original publication of this algorithm in [3], our group has continued work in this area. The “Snark” algorithm [11] improves on the algorithm presented here by not requiring “tagged” pointers. The “Hat Trick” algorithm [24] further improves on the Snark algorithm by allowing list nodes to be allocated in bulk and reused before being reclaimed, thereby significantly reducing the overhead of frequent allocation. All of these linked-list-based algorithms depend on garbage collection (GC) to reclaim linked-list nodes when they are no longer used. We have also shown how these algorithms can be transformed into equivalent ones that do not depend on garbage collection, using our Lock-Free Reference Counting (LFRC) methodology [12, 24].

1.2 The new algorithms

This paper presents two novel deque implementations that are non-blocking and linearizable, and do not suffer from the above-mentioned drawbacks of previous DCAS-based algorithms. The new array-based algorithm we present in Section 3 allows uninterrupted concurrent access to both ends of the deque, while returning appropriate exceptions in the tricky boundary cases when the deque is empty or full. The key to our algorithm is the rather interesting realization that a processor can detect these boundary cases, that is, determine whether the array is empty or full, without needing to check the relative locations of the two end pointers in one atomic operation.

In Section 4 we present a linked-list-based algorithm that does not restrict concurrency in accessing the deque’s two ends. The algorithm is based on a technique for “splitting” the pop operation into two steps, marking that a node is about to be deleted, and then deleting it. Once marked, the node is considered “deleted,” and the actual deletion from the list can then be performed by the next push or next pop operation on that side of the deque. The key to making this algorithm work is the use of DCAS to correctly synchronize delete operations when processors detect that there are only marked nodes in the list, and attempt to delete one or more of these nodes concurrently from both ends. The cost of this splitting technique is an extra DCAS per pop operation. The benefit is that it allows non-blocking completion without needing to synchronize on both of the deque’s end pointers with a DCAS. The splitting also requires allocating a bit in the pointer word to indicate if it is pointing to a marked node that needs to be deleted. However, this extra bit can be avoided, as we explain later, by adding two dummy “delete-bit” records to the structure.

In summary, we believe that through the design of linearizable lock-free implementations of classical data structures such as deques, we will be able to understand better the power of the DCAS abstraction, and whether one should continue the effort to provide support for implementing it [1, 5, 8, 9, 16, 17, 30] on concurrent hardware and software processing platforms. The next section presents our compu-

tation model and a formal specification of the deque data structure.

2 Modeling DCAS and Deques

Our paper deals with implementing a deque on a shared memory multiprocessor machine, using the DCAS operation. This section describes the computation model we assume and specifies the semantics of the deque data structure.

Our computation model follows [6, 7, 22]. A *concurrent system* consists of a collection of n processors.¹ Processors communicate through shared data structures called *objects*. Each object has a set of primitive *operations* that provide the only means of manipulating that object. Each processor P is a sequential thread of control [22] which applies a sequence of operations to objects by issuing an invocation and receiving the associated response. A *history* is a sequence of invocations and responses of some system execution. Each history induces a “real-time” order of operations where an operation A *precedes* another operation B if A ’s response occurs before B ’s invocation. Two operations are *concurrent* if neither precedes the other. A *sequential history* is a history in which each invocation is followed immediately by its corresponding response. The *sequential specification* of an object is the set of *legal* sequential histories associated with it. The basic correctness requirement for a concurrent implementation is *linearizability* [22]: every concurrent history is “equivalent” to some legal sequential history which is consistent with the real-time order induced by the concurrent history. In a linearizable implementation, operations appear to take effect atomically at some point between their invocation and response. In our model, a shared memory of a multiprocessor machine is a linearizable implementation of an object that provides every processor P_i with the following set of sequentially specified machine operations (see [20, 19] for details):

$Read_i(L)$ reads location L and returns its value.

$Write_i(L, v)$ writes the value v to location L .

$DCAS_i(L1, L2, o1, o2, n1, n2)$ is a double compare-and-swap operation with the semantics described in the next section.

We assume, based on current trends in computer architecture design [18], that DCAS is a relatively expensive operation, that is, has longer latency than traditional CAS, which in turn has longer latency than either a read or a write. We assume this is true even when operations are executed sequentially. We also assume the availability of a storage allocation/collection mechanism as in Lisp [31] and the Java™ programming language [15]. The details of the allocator are not exposed to the user², yet we assume that it includes an operation:

$New_i(v)$ that allocates a new structure v in memory and returns a pointer to it.

¹Of course, in real software systems, we would likely speak of our algorithms being executed by *threads* running on processors. This distinction is not important for the purpose of presenting our algorithms.

²Note that the problem of implementing a non-blocking storage allocator is not addressed in this paper but would need to be solved to produce a completely non-blocking deque implementation.

The implementations we present are *non-blocking* (also called *lock-free*) [19]. Let us use the term *higher-level operations* in referring to operations of the data type being implemented, and *lower-level operations* in referring to the (machine) operations in terms of which it is implemented. A *non-blocking* implementation is one in which any infinite history containing a higher-level operation that has an invocation but no response must also contain infinitely many responses concurrent with that operation. In other words, if some higher level operation continuously takes steps and does not complete, it must be because some other invoked operations are continuously completing their responses. This definition guarantees that the system as a whole makes progress and that individual processors cannot be blocked, only delayed by other processors continuously taking steps. Using locks would violate the above condition, hence the alternate name: *lock-free*.

2.1 The DCAS operation

Figure 1 defines the semantics of two forms of the DCAS operation, via pseudocode. These operations are assumed to be executed atomically, either through hardware support [21, 27, 28], through a non-blocking software emulation [8, 30], or via a blocking software emulation [2]. Note that there are two forms of the DCAS operation. One simply returns a boolean success/failure indication, while the other provides an atomic view of the two addressed memory locations. In the latter form, the third and fourth arguments are *pointers* to the “old” values, rather than the values themselves. In this case, the DCAS operation stores the atomic view of the two addressed locations in the locations pointed to by these pointers, whether the operation succeeds (i.e., the comparisons succeed and the writes occur) or fails. The form of each DCAS operation in our algorithms can be determined by examining the types of the third and fourth arguments. If they are pointers to values of the type contained in the addresses passed in the first two arguments, then the second form is used; otherwise the first form is used. It is worth noting that our algorithms depend on the stronger second form only for one optimization, as discussed in the algorithm description in Section 3; foregoing this optimization yields algorithms that can be implemented using only the weaker first form.

2.2 The Deque data structure

A *bounded deque* object S is a concurrent shared object created by a `make_deque(length_S)` operation, for $\text{length_S} \geq 1$, and allowing each processor P_i , $0 \leq i \leq n - 1$, to perform one of four types of operations on S : `pushRighti(v)`, `pushLefti(v)`, `popRighti()`, and `popLefti()`. Each push operation has input v of type `val`, and returns either “okay” or “full.” Each pop operation returns an output in the set `val`, or a special “empty” value.

We require that a concurrent implementation of a deque object be one that is linearizable to a standard sequential deque of the type described in [23]. We specify this sequential deque using a state-machine representation that captures all of its allowable sequential histories. These sequential histories consist of all sequences of push and pop operations that can be generated by execution of the state machine. In the following description, we abuse notation slightly for the sake of brevity.

The state of a deque is a sequence [10] of items $S = \langle v_0, \dots, v_k \rangle$ from the set `val`, having cardinality $0 \leq |S| \leq$

```

boolean DCAS(val *addr1, val *addr2,
             val old1, val old2,
             val new1, val new2) {
    atomically {
        if ((*addr1 == old1) && (*addr2 == old2)) {
            *addr1 = new1;
            *addr2 = new2;
            return true;
        } else {
            return false;
        }
    }
}

boolean DCAS(val *addr1, val *addr2,
             val *old1, val *old2,
             val new1, val new2) {
    atomically {
        if ((*addr1 == *old1) && (*addr2 == *old2)) {
            *addr1 = new1;
            *addr2 = new2;
            return true;
        } else {
            *old1 = *addr1;
            *old2 = *addr2;
            return false;
        }
    }
}

```

Figure 1: The Double Compare-and-Swap operation.

`length_S`. The deque is initially in the empty state (following `make_deque(length_S)`), that is, has cardinality 0, and is said to have reached a full state if its cardinality is `length_S`.

The four possible push and pop operations, executed sequentially, induce the following state transitions of the sequence $S = \langle v_0, \dots, v_k \rangle$, with appropriate returned values:

- `pushRight(v_{new})` if S is not full, changes S to be the sequence $S = \langle v_0, \dots, v_k, v_{new} \rangle$ and returns “okay”; if S is full, it returns “full” and S is unchanged.
- `pushLeft(v_{new})` if S is not full, changes S to be the sequence $S = \langle v_{new}, v_0, \dots, v_k \rangle$ and returns “okay”; if S is full, it returns “full” and S is unchanged.
- `popRight()` if S is not empty, changes S to be the sequence $S = \langle v_0, \dots, v_{k-1} \rangle$ and returns v_k ; if S is empty, it returns “empty” and S is unchanged.
- `popLeft()` if S is not empty, changes S to be the sequence $S = \langle v_1, \dots, v_k \rangle$ and returns v_0 ; if S is empty, it returns “empty” and S is unchanged.

For example, starting with an empty deque $S = \langle \rangle$, the following sequence of operations and corresponding transitions can occur: `pushRight(1)` changes the state to $S = \langle 1 \rangle$; `pushLeft(2)` transitions to $S = \langle 2, 1 \rangle$; a subsequent `pushRight(3)` transitions to $S = \langle 2, 1, 3 \rangle$. A subsequent `popLeft()` transitions to $S = \langle 1, 3 \rangle$ and returns 2. Another subsequent `popLeft()` transitions to $S = \langle 3 \rangle$ and returns 1 (which had been pushed from the right).

An *unbounded deque* object has almost exactly the same specification; the differences are that the `make_deque` operation takes no length specification, and the push operations never return “full.”³ In sections describing the array-based

³We prove that our linked-list-based implementation meets the unbounded deque specification under the assumption that memory allocator never fails. In the actual implementation, the push operations return “full” in the case that the memory allocator fails.

and link-list-based implementations, “deque” refers respectively to bounded and unbounded deques.

3 The array-based algorithm

```
Initially L == 0, R == 1 mod length_S-1;
        S[0..length_S-1] of "null";

0 val popRight {
1   newS = "null";
2   while (true) {
3     oldR = R;
4     newR = (oldR - 1) mod length_S;
5     oldS = S[newR];
6     if (oldS == "null") {
7       if (oldR == R)
8         if (DCAS(&R, &S[newR],
9                 oldR, oldS, oldR, oldS))
10          return "empty";
11    }
12    else {
13      saveR = oldR;
14      if (DCAS(&R, &S[newR],
15              &oldR, &oldS, newR, newS))
16        return oldS;
17    }
18    else if (oldR == saveR) {
19      if (oldS == "null") return "empty";
20    }
21  }
22 }
```

Figure 2: The array-based deque - right-hand-side pop.

```
1 val pushRight(val v) {
2   while (true) {
3     oldR = R;
4     newR = (oldR + 1) mod length_S;
5     oldS = S[oldR];
6     if (oldS != "null") {
7       if (oldR == R)
8         if (DCAS(&R, &S[oldR],
9                 oldR, oldS, oldR, oldS))
10          return "full";
11    }
12    else {
13      saveR = oldR;
14      if (DCAS(&R, &S[oldR],
15              &oldR, &oldS, newR, v))
16        return "okay";
17    }
18    else if (oldR == saveR)
19      return "full";
20  }
21 }
```

Figure 3: The array-based deque - right-hand-side push.

The following is a non-blocking implementation of a deque in an array using DCAS. We describe in detail the code in Figures 2 and 3, which deal with the right-hand-side push and pop operations, with the understanding that the left-hand-side operations in Figures 30 and 31 are symmetric. As depicted in Figure 4, the deque consists of an array $S[0..length_S-1]$ indexed by two counters, R and L . We assume that mod is the modulus operation over the integers ($-1 \text{ mod } 6 = 5$, $-2 \text{ mod } 6 = 4$, and so on). Henceforth, we assume that all indexes into the array S are modulo $length_S$, which implies that S is viewed as being circular. We also as-

sume a distinguished “null” value (denoted as “0” in some of the figures) that is not a member of the set val .

Think of the array $S[0..length_S-1]$ as if it were laid out with indexes increasing from left to right. The code works as follows. Initially, L points immediately to the left of R . (That is, $(L + 1) \text{ mod } length_S = R$. The pointers L and R always point to the next location (if any) into which a value can be inserted from the left or right, respectively. If there is no value immediately to the right of L (respectively, to the left of R), then the deque is in the empty state. Similarly, if there is a non-null value in location L (respectively, location R), then the deque is in the full state. Figure 4 shows such empty and full states.

To perform a `popRight`, a processor first reads R and the location in S corresponding to $R-1$ (Lines 3-5). It then checks whether $S[R-1]$ is null. (As noted above, $S[R-1]$ is shorthand for $S[R-1 \text{ mod } length_S]$). If $S[R-1]$ is null, then the processor reads R again to see if it has changed (Lines 6-7). This additional read is a performance enhancement added under the assumption that the common case is that a null value is read because another processor “stole” the item, and not because the deque is really empty. The test is thus that if R has not changed and $S[R-1]$ is null, then the deque may be empty because the location to the left of R always contains a value unless there are no items in the deque. However, the conclusion that the deque is empty can only be made based on an instantaneous view of R and $S[R-1]$, and so the processor performs a DCAS to check if this is in fact the case (Lines 8-10). If so, the processor returns an indication that the deque is empty. If not, then either the value read in $S[R-1]$ is no longer null or the index R has changed. In either case, the processor must loop around and start again, because there might now be an item to pop.

If $S[R-1]$ is not null, the processor attempts to pop that item (Lines 12-20). It uses a DCAS to try to decrement the counter R and to place a null value in $S[R-1]$, while returning (via `oldS` and `oldR`) the old value in $S[R-1]$ and the old value of the counter R (Lines 13-15). A successful DCAS (and hence a successful `popRight` operation) is depicted in Figure 5. (In this and later figures, solid boxes indicate variables accessed by the DCAS under consideration.) If the DCAS succeeds, the processor returns the value that was in $S[R-1]$ before the DCAS. If it fails, then it needs to check what the reason for the failure was. If the reason for the DCAS failing was that R changed, then the processor must retry (repeat the loop) because there may be items still left in the deque. If the value of R returned in `oldR` is the same as the expected value `saveR` passed to the DCAS (Line 17), then the DCAS failed because $S[R-1]$ contained a value other than the expected value. The value of $S[R-1]$ observed by the DCAS is returned via the variable `oldS`. If

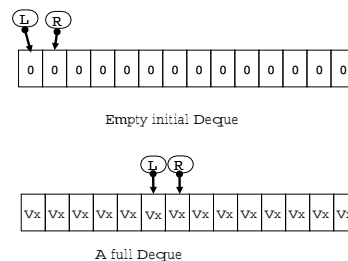


Figure 4: Empty and full array-based deques.

this value is null (Line 18), then the deque was empty when the DCAS was executed. This empty deque would have been the result of a competing `popLeft` that “stole” the last item from the `popRight`, as in Figure 6. Otherwise, `oldS` is neither the expected value nor null. This implies that some other operations succeeded in updating `S[R-1]`. Such updates can be achieved only by DCAS’s of completing operations; thus lock-freedom is not compromised by retrying in this case.

We note that the algorithm would be still be correct if line 7, and/or lines 17 and 18, were deleted. We have included these lines because we believe they may be optimizations for common usage patterns. If the comparison at line 7 fails, then we avoid a presumably-costly DCAS that would likely fail. Lines 17-18 can detect that the deque was empty when the previous DCAS failed, and can therefore return immediately without retrying. On the other hand, eliminating lines 17-18 yields an algorithm that does not require the stronger version of DCAS that returns an atomic view of the contents of the two addresses even on failure. While both of these code fragments may avoid overhead in some cases, there is also overhead associated with including them. Experimentation would be required to determine whether either or both of these code fragments should be included for a specific application and system context.

To perform a `pushRight`, a sequence similar to `popRight` is performed. As depicted in Figure 7, for a given `R`, the location into which an item will be pushed is the one to which `R` points (Line 5). A successful `pushRight` operation into an empty deque is depicted in Figure 7, and a successful `pushRight` operation into an almost-full deque appears in the bottom of Figure 8. All tests to see if a location is null are now replaced with tests to see if it is non-null (such as in Lines 6-10). Furthermore, in the final stage of the code, in case the DCAS failed, there is a check to see if the `R` index has changed. If it has not, then the failure must be due to a non-null value in that location, which means that the deque is full. Unlike the case of a `popRight`, the DCAS (Lines 14-15) is only checking to see if there was a null value, and if none is found it does not matter what the non-null value was: in all cases it means that the deque is full (Line 18).

Figure 8 shows how an almost-full deque becomes full following push operations from the left and the right. Notice how `L` has wrapped around and is “to-the-right” of `R`, until the deque becomes full, in which case `L` and `R` cross again. This switching around of the relative location of the `L` and `R` pointers is rather confusing. The key to our algorithm is the observation that we can determine the state of the deque,

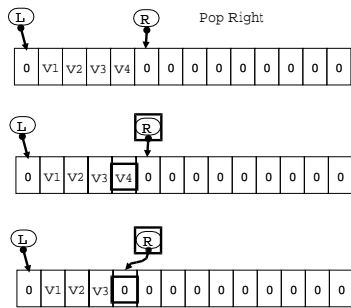


Figure 5: A successful array-based `popRight`.

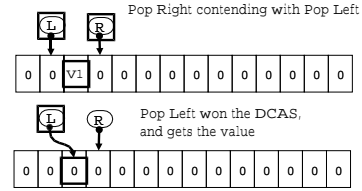


Figure 6: A `popRight` contending with a `popLeft`.

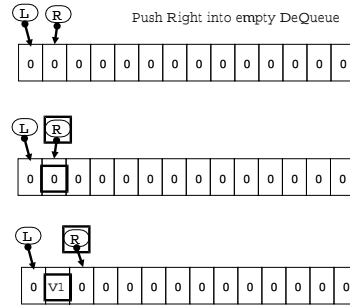


Figure 7: A successful array-based `pushRight`.

not based on these relative locations of `L` and `R`, but rather by examining the combination of where a given pointer variable is pointing and the value in the location associated with that pointer. Section 5.1 presents an overview of a proof, which we accomplished with the aid of a mechanical theorem prover, of the following theorem:

Theorem 3.1 *The array based deque algorithm of Section 3 is a non-blocking linearizable implementation of a deque data structure.*

4 The linked-list-based algorithm

The previous sections presented an array-based deque implementation appropriate in situations where the maximum size of the deque can be predicted in advance (and allocating in advance space sufficient to accommodate the maximum size is acceptable). In this section we present an implementation that avoids the need to limit size, by allowing dynamic

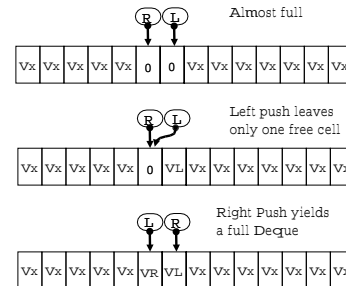
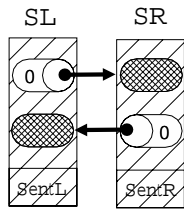
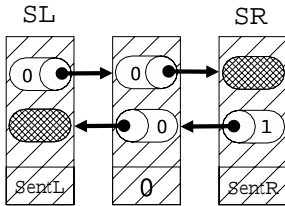


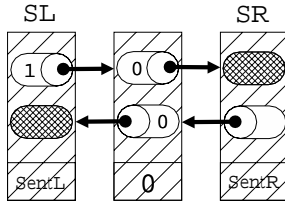
Figure 8: Filling the array.



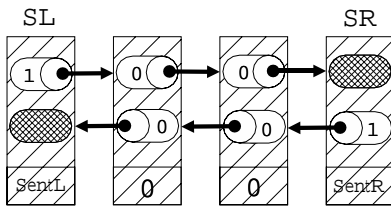
Empty Deque



Empty Deque with a right-deleted cell



Empty Deque with a left-deleted cell



Empty Deque with two deleted cells

```
typedef node {
    pointer *L;
    pointer *R;
    val_or_null_or_sentL_or_sentR value;
}
```

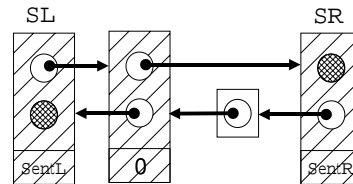
It is assumed that there are three distinguished values (called `null`, `sentL`, and `sentR`) that can be stored in the value field of a node but are never requested to be pushed onto the deque. The doubly-linked list has two distinguished nodes called “sentinels.” The left sentinel is at a known fixed address `SL`; its `L` pointer is never used, and its value field always contains `sentL`. Similarly, the right sentinel is at a known fixed address `SR` with value `sentR` and an unused `R` pointer.

The basic intuition behind this implementation is that a node is always removed (as part of a `popRight` or `popLeft` operation) in two separate, atomic steps: first the node is “logically” deleted, by replacing its value with `null` and setting a special `deleted` bit to 1 in the sentinel to indicate the presence of a logically deleted node; second, the node is “physically” deleted by modifying pointers so that the node is no longer in the doubly-linked chain of nodes, as well as resetting the bit in the sentinel to 0. If a processor that is removing a node is suspended between these two steps, then any other process can perform the second step or otherwise work around the fact that the second step has not yet been performed.

More concretely, the `deleted` bit in each sentinel is represented as part of its pointer into the list. The following structure is thus maintained in a single word, by assuming sufficient pointer alignment to free one low-order bit.⁴

```
typedef pointer {
    node *ptr;
    boolean deleted;
}
```

Initially `SR->L == SL` and `SL->R == SR`, as in the top part of Figure 9. All push and pop procedures use an auxiliary delete procedure, which we describe last. We describe only the `popRight` and `pushRight` procedures. The `popLeft` and `pushLeft` procedures are symmetric and can be found in Figures 32 and 33.



Empty Deque with one deleted cell marked by a right dummy node

Figure 9: Four versions of an empty linked-list-based deque.

memory allocation. We represent a deque as a doubly-linked list. Each node in the list contains two link pointers and a value.

Figure 10: Replacing the deleted bit with a dummy node.

⁴One can altogether eliminate the need for a “deleted” bit by introducing a special dummy type “delete-bit” node, distinguishable from regular nodes, in place of the bit. Each processor would have a dummy node for the left and one for the right, and pointing to a node indirectly via its dummy node, as in Figure 10, represents a bit value of true, and pointing directly represents false.

The code for the `popRight` procedure appears in Figure 11. The processor begins by checking if the right sentinel is in one of the following states:

- it points to the left sentinel (Line 5 of the code and the top of Figure 9),
- the deleted bit of its left pointer is set (Line 6), so that the node this pointer references is logically deleted, and right-side operations must complete the physical deletion (Line 7) before retrying the loop,
- it points to a node that has a null value (and is thus logically deleted) (Lines 8-12 of the code and the upper part of Figure 15 and the three lower parts of Figure 9), or,
- it points to a node that has a non-null value (Lines 13-19 of the code and Figure 12).

If the right sentinel points directly to the left sentinel (Line 5), then the deque is empty, as in the top of Figure 9, and the processor can return “empty.” Otherwise, the processor checks whether the `deleted` bit in the right sentinel is true (Line 6). If so, the processor calls the `deleteRight` procedure, which ensures that the null node on the right-hand side is removed, and then retries the pop (Line 7). If the `deleted` bit (in the right sentinel) was observed to be false, then the processor checks whether the node to be popped has a null value (Line 8). If it does, and the right sentinel’s left pointer, including its false deleted bit, has not changed since the read at Line 3, then the deque is empty (as depicted in the third diagram from the top of Figure 9), because the right sentinel points to a node deleted by a `popLeft` operation. The DCAS at Line 9 performs an atomically check confirming that the pointer and deleted bit, and the null value, have not changed.⁵ If the DCAS fails, then the deque must have been modified between the first read and the DCAS, so the processor retries.

```

1 val popRight() {
2   while (true) {
3     oldL = SR->L;
4     v = oldL.ptr->value;
5     if (v == "sentL") return "empty";
6     if (oldL.deleted == true)
7       deleteRight();
8     else if (v == "null") {
9       if (DCAS(&SR->L, &oldL.ptr->value,
10              oldL, v, oldL, v))
11         return "empty";
12     }
13     else {
14       newL.ptr = oldL.ptr;
15       newL.deleted = true;
16       if (DCAS(&SR->L, &oldL.ptr->value,
17              oldL, v, newL, "null"))
18         return v;
19     }
20   }
21 }

```

Figure 11: The linked-list-based deque - right side pop.

Finally, there is the case in which the `deleted` bit is false and `v` is not null, as depicted in Figure 12. Using a DCAS,

⁵While completing the proof, we noticed that, because “null” values are stable, only the pointer and deleted bit must be confirmed, and that this confirmation can be achieved with a single read of `SR->L`.

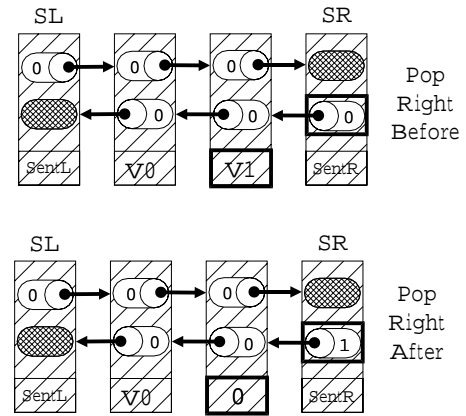


Figure 12: Non-null node before and after being popped by a `popRight`.

```

1 val pushRight(val v) {
2   newL.ptr = new Node();
3   if (newL.ptr == null) return "full";
4   newL.deleted = false;
5   while (true) {
6     oldL = SR->L;
7     if (oldL.deleted == true)
8       deleteRight();
9     else {
10      newL.ptr->R.ptr = SR;
11      newL.ptr->R.deleted = false;
12      newL.ptr->L = oldL;
13      newL->value = v;
14      oldLR.ptr = SR;
15      oldLR.deleted = false;
16      if (DCAS(&SR->L, &oldL.ptr->R,
17             oldL, oldLR, newL, newL))
18        return "okay";
19    }
20  }
21 }

```

Figure 13: The linked-list-based deque - right side push.

the processor atomically swaps `v` out from the node, changing the value to null, and at the same time changing the `deleted` bit in the pointer to it in `SR` to true (Lines 14-17). If the DCAS is successful (Line 18), the processor returns `v` as the result of the pop, leaving the deque in a state where the right sentinel’s `deleted` bit is true, indicating that the node is logically deleted. The next `popRight` or `pushRight` will call the `deleteRight` procedure,⁶ to complete the physical deletion. If the DCAS fails, then concurrent operations must have modified the sentinel pointer. The processor therefore retries.

The code for the `pushRight` procedure appears in Figure 13. The processor starts, as depicted in Figure 14, by allocating a new node pointed to by a variable `newL` (Lines 2-4). If the allocation fails, a “full” indicator is returned. Otherwise, the processor checks if the `deleted` bit in the right sentinel is true (Lines 6-7). If so, as depicted in Figure 15, it calls (in Line 8) the `deleteRight` procedure, which ensures the removal of the null node to which the sentinel points, prior to continuing with the push. If the `deleted` bit is false, then the processor fills in the pointers in the `newL` node to point to the right sentinel and left neighbor of that

⁶Note that the `popRight` operation could also call the `deleteRight` procedure before returning `v`.

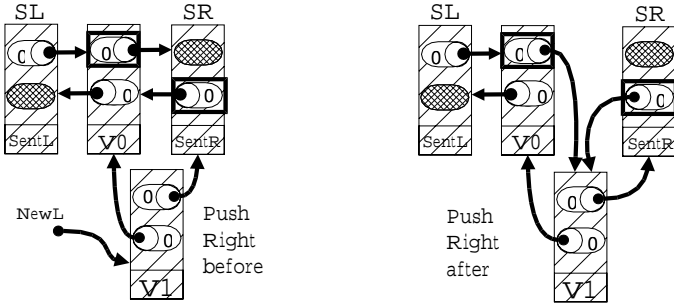


Figure 14: Before `pushRight` (left) and after a successful `pushRight` (right).

sentinel (Lines 10-13).

The processor then tries to insert the new node into the list by using a DCAS to redirect the pointers in the sentinel and its left neighbor to the new node, as depicted in Figure 14 (Lines 14-17). If the DCAS is successful, the node is added (Line 18), and otherwise it must be the case that the deque was changed concurrently, and so the processor must loop and try to push the node again.

The code of the `deleteRight` procedure appears in Figure 17. The procedure begins by checking that the left pointer of the right sentinel has its `deleted` bit set to true (Lines 3-4). If not, the deletion has been completed by another processor and the current processor can return. If the bit is true, the next step is to determine the configuration of the deque. To this end, the processor reads the left pointer of the node to be deleted (Line 5). At Line 6, the processor reads the value field of the node to which this pointer refers, which may contain a null value, a non-null value, or the `sentL` value. The actions to be taken for the non-null value, as shown in the upper part of Figure 15, and for the `sentL` value, as shown in the second part of Figure 9, are the same (Lines 6-13). The processor checks whether `oldL.ptr` (previously read from `SR->L`) and `oldLLR.ptr` (the right pointer, as read at Line 7, of the left neighbor of the node to be deleted) point to the same node (Line 8), the one to be deleted. If this is not the case, the processor retries because the deque has been modified. If they do point to the same node (Lines 9-13), then the processor uses a DCAS to attempt to redirect the pointers so that the right sentinel and the node pointed to by `oldLL` point to each other, thereby deleting the null node pointed to by `oldL`. The case of the null value is somewhat different. A null value means that the deque consist of just two non-sentinel nodes, each of which is deleted (i.e., the state shown in the bottom part of Figure 9). To deal with this possibility, the processor checks whether the `deleted` bit of the left sentinel is true (Lines 17-18). If so, it attempts to point the sentinels to each other using a DCAS (Lines 23-24). In case the DCAS fails, the processor must go through the loop again, until the deletion is completed, i.e. the right sentinel's `deleted` bit is observed to be false.

The most interesting case occurs when the deque contains only two nodes, `left` and `right`, both logically deleted and thus containing the null value, and a `deleteLeft` is about to be executed concurrently with a `deleteRight`, as depicted in the top part of Figure 16. This state can be reached by the following sequence of events. The `deleteLeft`

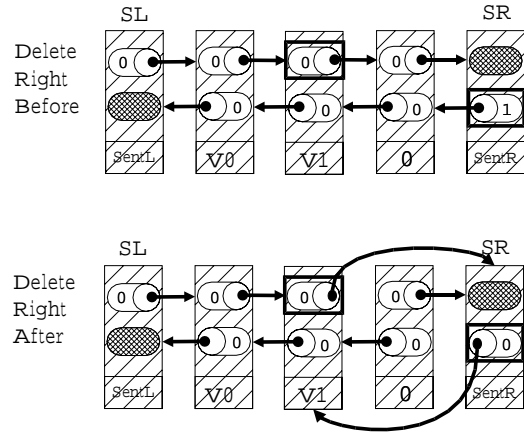


Figure 15: Null node before and after being deleted by a `deleteRight`.

operation started first, while the `right` node was still non-null. The `deleteLeft` then attempted to remove the `left` node by performing a DCAS on the left sentinel's pointer and the `right` node's L pointer. The `deleteRight`, which started later, detected the two null nodes and attempted to remove both, by using a DCAS to redirect the pointers of the sentinels towards each other (Lines 17-25). As can be seen at the top of Figure 16, the DCAS operations performed by `deleteLeft` and `deleteRight` overlap on the pointer in the left sentinel. If the `deleteLeft` executes the DCAS first, the result of the success is a deque with one null node and the `deleted` bit of the right sentinel still true, as shown in the bottom left of Figure 16. If, on the other hand, the `deleteRight` executes the DCAS first, the result is an empty deque consisting only of the two sentinels, with their `deleted` bits false, as in the bottom right of Figure 16.

Section 5.2 presents an overview of a proof, which we accomplished with the aid of a mechanical theorem prover, of the following theorem:

Theorem 4.1 *The linked list based deque algorithm of Section 4 is a non-blocking linearizable implementation of a deque data structure.*

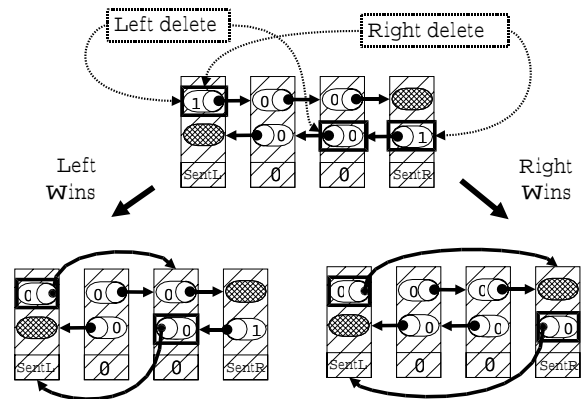


Figure 16: Contending `deleteLeft` and `deleteRight`.

```

1 deleteRight() {
2   while (true) {
3     oldL = SR->L;
4     if (oldL.deleted == false) return;
5     oldLL = oldL.ptr->L.ptr;
6     if (oldLL->value != "null") {
7       oldLLR = oldLL->R;
8       if (oldL.ptr == oldLLR.ptr) {
9         newR.ptr = SR;
10        newR.deleted = false;
11        if (DCAS(&SR->L, &oldLL->R,
12              oldL, oldLLR, oldLL, newR))
13          return;
14      }
15    }
16    else { /* there are two null items */
17      oldR = SL->R;
18      if (oldR.deleted) {
19        newL.ptr = SL;
20        newL.deleted = false;
21        newR.ptr = SR;
22        newR.deleted = false;
23        if (DCAS(&SR->L, &SL->R,
24              oldL, oldR, newL, newR))
25          return;
26      }
27    }
28  }
29 }

```

Figure 17: The linked-list-based deque - right side delete.

5 Proofs

In this section we describe how we verified the two deque algorithms presented in this paper with the aid of a mechanical theorem prover. The two proofs have much in common; here we give a general overview of the proof techniques before discussing the individual proofs.

To prove linearizability, we define a single *linearization point* for each operation execution. Intuitively, the operation can be considered to take effect atomically at this point. In our linearizability proofs, each linearization point corresponds to one of the atomic steps of the process executing the operation. Therefore, the linearization points of an execution define a total order that extends the real-time partial order over operation executions. Our proof obligation is that the values returned by the operations are consistent with a sequential execution of the operations in this order.

To achieve this, we define an abstraction function A mapping states of the implementation to abstract deque values, and show that the state transition at each linearization point corresponds to a correct application of the linearized operation to the abstract deque. (That is, if the transition occurring at the linearization point takes the implementation from state S_0 to state S_1 , then $A(S_1)$ is the result of applying the linearized operation to $A(S_0)$.) We also show that *only* transitions that are linearization points change the abstract value of the deque. “Tracking” the abstract value in this way allows us to determine, for each linearization point, the correct return value of the linearized operation.

To aid in these proofs, we define a representation invariant R that holds in all reachable implementation states. We prove that R is an invariant of the implementation by proving that R holds initially and that all state transitions preserve R . This invariant allows us to avoid further reasoning about interleavings of concurrent operations because we know that it holds immediately before each linearization point. (It also defines the domain of the abstraction function

A.)

We have performed these proofs with the aid of a mechanical theorem prover. We used the *Simplify* [14] prover, largely because one of the authors was a developer of that system. This prover accepts first-order formulas, and has built-in support for linear inequalities over integer variables. In contrast with some other proof checkers, it requires relatively little user interaction to guide proofs (though occasionally this extra automation leads to failed proofs). We claim no special distinction for this theorem prover; we believe the same style of proof would have been feasible in several systems.

In our experience, the difficult part of a correctness proof of a concurrent algorithm concerns the interactions between concurrent processes, and not the sequential aspects of individual processes. We have therefore focussed our efforts with the mechanical prover on the concurrent aspects, performing the simpler sequential reasoning by hand. Of course, this reasoning could also be mechanized; indeed, this is quite similar to the “verification condition generation” performed by program verification systems such as the ESC[13] system for which the Simplify prover was developed.

Our representation invariants are defined only in terms of shared variables (i.e., they do not refer to processes’ local variables or program counters). Therefore, in order to prove that they are invariant, we need only consider state transitions in which the value of some shared variable changes. In our algorithms, each such transition is the result of a DCAS execution.⁷ DCAS’s that fail do not affect any shared variables. Therefore we prove that, for any state that satisfies R and the preconditions required for the DCAS to succeed, R holds in the state reached by executing the DCAS.

In order to define the sequential semantics of dequeues, we axiomatize dequeues using *EmptyQ*, *singleton*, and *concat* constructors. We define function symbols *pushR*, *pushL*, *popR*, and *popL* to express the transformations performed by the abstract deque operations, and observer functions *peekR* and *peekL* to determine appropriate return values for “pop” operations. These axioms (shown in Figure 35) are used in both proofs.

5.1 Proof of the array-based implementation

We first consider the array-based implementation, and start by presenting its representation invariant, shown in Figure 18. We present the invariant using the syntax with which it is presented to the theorem prover, so a brief explanation of that syntax is in order. Like the LISP programming language, the Simplify input language uses S-expressions and prefix operator notation. This syntax is for the most part self-explanatory, with some exceptions:

- The (DEFPRED (**pred** *arg* ...) *body*) form defines **pred** to be a new predicate symbol with the given arguments; an application of **pred** is equivalent to *body* with actual arguments substituted for free occurrences of formals such as *arg*.
- The (BG_PUSH **pred**) form causes the prover to assume **pred** as an axiom. Typically, **pred** is a universally quantified formula.

⁷We do not consider fields of a newly-allocated heap object to be shared variables until a pointer to the object has been stored in some shared variable, because the object can only be accessed by the allocating thread.

- The (LBL label pred) form is semantically equivalent to `pred`; the label provides documentation, and aids in debugging failed proofs.
- The prover has built-in support for a theory of *maps*. The `select` function takes a map and an index, and returns the value stored in that map at that index. The form `(store m i v)` yields a map that is like the map `m`, except that it has value `v` at index `i`.
- Universally quantified formulas sometimes contain a (PATS pat ...) form. These have no semantic meaning; they are hints to the prover controlling when such formulas are instantiated in proofs.

The invariant is defined over three shared state variables: indices `L` and `R` into an array `S`, and that array itself. The predicate definition `RepInv` takes four arguments corresponding, respectively, to these three shared variables and the size of the array.

The first three conjuncts of the representation invariant shown in Figure 18 say that the array size is non-zero, and that the indices are within the range of the array. The fourth conjunct is a disjunction of three labeled predicates, each of which corresponds to a possible way that values can be arranged in the array. Intuitively, the non-null elements of the array form a contiguous sequence. However, because we represent the deque in a bounded array, this sequence can in fact “wrap around” past the end boundary. The invariant expresses the wrapped and non-wrapped cases explicitly. The case of a full deque is an additional possibility; a full deque may be either wrapped or non-wrapped, but it must be distinguished from an empty deque. Each of these three disjuncts specifies which array elements are null and which are non-null as a function of the `l` and `r` indices. For example, consider the disjunct labeled `FullQueue`. In a full deque, the `r` index is one more than the `l` index, modulo the size of the deque, as stated by the first conjunct. (This modulo is expressed with the `mod1` function, which is axiomatized in the full proof script.) However, as discussed in Section 3, this is also true of an empty deque. Therefore, the second conjunct is required to distinguish these two cases; this conjunct states that *all* of the values stored in the array are non-null. (We use zero to represent the null value.) The other two disjuncts express, for non-full deques, which values are null and which are non-null in the wrapped and non-wrapped cases described above.

Now we consider the abstraction function for the array-based implementation. This definition relies on a utility function: `AbsFuncContig` takes a map representing an array, and two indices. It returns the term representing the deque containing the values in the map from the first index, up to but not including the second index. Figure 19 presents the axiomatization of this function. (The actual axioms used in the proof script are semantically equivalent, but slightly modified to control the path of the proof.)

The abstraction function is defined by four mutually-exclusive cases: full, empty, non-wrapped, and wrapped. In the non-wrapped case, the abstract deque value is obtained (using `AbsFuncContig`) from the sequence of values between `l` and `r`. In the wrapped and full cases, the value is the concatenation of two abstract values for the two non-null subsequences of values in the array. In the empty case, there are no non-null values in the array; the abstract deque value is defined to be the `EmptyQ` constructor. Figure 20 shows these four definitions.

We can now present some of the verification conditions, i.e., the predicates representing the proof obligations that

```
(DEFPRED (RepInv l r s n)
  (AND
    (LBL PhysQueueSize (> n 0))
    (LBL RInRange (AND (<= 0 r) (< r n)))
    (LBL LInRange (AND (<= 0 l) (< l n)))
    (OR
      (LBL FullQueue
        (AND (EQ r (mod1 (+ l 1) n))
          (FORALL (i)
            (IMPLIES
              (AND (<= 0 i) (< i n))
              (NEQ (select s i) 0))))
        )
      (LBL Wrapped
        (AND (<= r l)
          (FORALL (i)
            (IMPLIES
              (AND (<= 0 i) (< i n))
              (IFF (AND (>= i r) (<= i l))
                (EQ (select s i) 0))))
          )))
    (LBL NotWrapped
      (AND (> r l)
        (FORALL (i)
          (IMPLIES
            (AND (<= 0 i) (< i n))
            (IFF (AND (> i l) (< i r))
              (NEQ (select s i) 0))))
          )))
    )))
```

Figure 18: Definition of representation invariant for the array-based implementation.

```
(BG_PUSH
  (FORALL (s i)
    (EQ (AbsFuncContig s i i) EmptyQ))

  (FORALL (i j s)
    (PATS (AbsFuncContig s i j))
    (IMPLIES (< i j)
      (EQ (AbsFuncContig s i j)
        (pushR (AbsFuncContig s i (- j 1))
          (select s (- j 1)))))))
)
```

Figure 19: Definition of `AbsFuncContig` for the array-based implementation.

must be discharged at each shared state transition and/or linearization point (some, but not all, transitions may be both of these.) As discussed previously, we must show preservation of the representation invariant at every shared state transition, and a correct abstract transition and return value at every linearization point. For convenience, when a transition is both a shared state transition and a linearization point, we combine all proof obligations into one predicate, because each proof obligation depends on the same preconditions for the transition.

We first consider the initial state. Figure 21 shows a predicate expressing that any array of length at least one, with all elements null, and with `L` and `R` indices 0 and 1 (modulo the array length), respectively, satisfies the representation invariant and represents an empty deque.

Next we consider the `popRight` operation, shown in Figure 2. This routine has three `return` statements. The first two, at lines 10 and 16, are executed only if preceding `DCAS` invocations (at lines 8 and 14, respectively) succeed; in each case the operation’s linearization point is defined to be the point at which the `DCAS` succeeds. `popRight` operations

```

(BG_PUSH
;; Full case
(FORALL (s l r n)
  (PATS (AbsFunc s l r n)
    (IMPLIES (AND (NEQ (select s l) 0)
      (EQ r (mod1 (+ l 1) n)))
      (EQ (AbsFunc s l r n)
        (concat (AbsFuncContig s r n)
          (AbsFuncContig s 0 r)
        )))))
;; Empty case
(FORALL (s l r n)
  (PATS (AbsFunc s l r n)
    (IMPLIES (AND (EQ (select s l) 0)
      (EQ r (mod1 (+ l 1) n)))
      (EQ (AbsFunc s l r n) EmptyQ)))
;; Non-wrapped.
(FORALL (s r l n)
  (PATS (AbsFunc s l r n)
    (IMPLIES
      (AND (NEQ r (mod1 (+ l 1) n))
        (> r l))
      (EQ (AbsFunc s l r n) (AbsFuncContig s (+ l 1) r))))
;; Wrapped case.
(FORALL (s r l n)
  (PATS (AbsFunc s l r n)
    (IMPLIES
      (AND (NEQ r (mod1 (+ l 1) n))
        (<= r l))
      (EQ (AbsFunc s l r n)
        (concat (AbsFuncContig s (+ l 1) n)
          (AbsFuncContig s 0 r))))))
)

```

Figure 20: Definition of abstraction function for the array-based implementation.

```

(FORALL (S n)
  (IMPLIES
    (AND
      (>= n 1)
      (FORALL (i)
        (IMPLIES
          (AND (<= 0 i) (< i n))
          (EQ (select S i) 0)))
      )
    (AND
      (RepInv 0 (mod1 1 n) S n)
      (EQ (AbsFunc S 0 (mod1 1 n) n) EmptyQ)
    )))

```

Figure 21: Verification of initial conditions for array-based implementation.

that return from line 18 are linearized to the point at which the preceding DCAS failed (at line 14). (In this case, the atomic view of a pair of memory locations provided by the failure of the DCAS at line 14 establishes that the deque was empty when the preceding DCAS failed.)

Figure 22 shows the verification conditions for the return statements at lines 10 and 16. The first of these predicates says that in a memory state that satisfies the representation invariant, if the DCAS “confirms” that the value to the (logical) left of R is null, then the corresponding abstract deque value is the empty deque, justifying the “empty” outcome at this linearization point. This DCAS performs no state change even if it succeeds (because its new value arguments are the same as its old value arguments), and therefore does not falsify the invariant.

```

;; popRight: DCAS at line 8 successful; return at line 10
(FORALL (r l s n)
  (IMPLIES (AND
    (RepInv l r s n)
    (EQ (select s (mod1 (- r 1) n)) 0))
    (EQ (AbsFunc s l r n) EmptyQ)))
)
;; popRight: DCAS at line 14 successful; return at line 16
(FORALL (r l s n)
  (IMPLIES
    (AND (RepInv l r s n)
      (NEQ (select s (mod1 (- r 1) n)) 0)
    )
    (AND
      (LBL RepInvPreserved
        (RepInv l (mod1 (- r 1) n) (store s (mod1 (- r 1) n) 0) n)
      )
      (LBL QNotEmpty
        (NEQ (AbsFunc s l r n) EmptyQ))
      (LBL ProperTransition
        (EQ (AbsFunc (store s (mod1 (- r 1) n) 0)
          l (mod1 (- r 1) n) n)
          (popR (AbsFunc s l r n))))
      (LBL ProperReturnVal
        (EQ (peekR (AbsFunc s l r n))
          (select s (mod1 (- r 1) n))))
    )
  )))

```

Figure 22: Verification conditions for array `popRight`, Figure2.

The proof obligations associated with the DCAS at line 14 are more complicated, because this DCAS does update the state. The predicate states that if the representation invariant holds before the update, and the DCAS confirms that the value to be returned is non-null, then four properties hold. First, the representation invariant holds in the state immediately after the execution of the DCAS. This state is represented in terms of the state before the DCAS: the post-DCAS value of R is one less than the pre-DCAS value (modulo the physical deque size), and the post-DCAS array is the pre-DCAS array with the zero/null value stored at the post-DCAS value of R . Second, the precondition of an abstract pop operation that returns a value (instead of “empty”) requires that the deque be non-empty before the operation. Third, the abstract deque value makes the appropriate transition: the abstract value after the transition is equal to the result of applying the `popR` to the abstract value before the transition. Finally, the value returned by the `popRight` operation is correct.

The last case, where the “empty” value is returned in line 18 after a failed DCAS, turns out to have exactly the same verification condition as the DCAS at Line 8.

The `pushRight` operation is fairly similar to `popRight`. Again, it has three return statements, two that follow successful DCAS invocations, and one following an unsuccessful invocation. Much as two of the outcomes for `popRight` determined that the deque was empty, and the other performed the operation, two of the outcomes for `pushRight` determine that the deque is full, and the other performs the operation.

The verification conditions for the left-side operations are exactly analogous to those for the right-side operations.

Several lemmas were required to allow the theorem prover to discharge the proof obligations outlined above. All of these lemmas elucidate properties of the `AbsFuncContig` function. Figure 23 shows these lemmas. The first says that a store to the S array outside the index range given in an application of `AbsFuncContig` does not affect the value of

```

(FORALL (s l r i v)
  (IMPLIES (AND (>= r l) (OR (< i l) (>= i r)))
    (EQ (AbsFuncContig (store s i v) l r)
      (AbsFuncContig s l r))))

(FORALL (s l r)
  (PATS (len (AbsFuncContig s l r)))
  (IMPLIES (>= r l)
    (EQ (len (AbsFuncContig s l r)) (- r l))))

(FORALL (i j s)
  (PATS (AbsFuncContig s i j))
  (IMPLIES (< i j)
    (EQ (AbsFuncContig s i j)
      (pushL (AbsFuncContigRest s (+ i 1) j)
        (select s i)))))

```

Figure 23: Lemmas needed for array-based implementation.

that application. The second shows how to compute the length of an `AbsFuncContig` application. The last lemma addresses the one way in which the left-side and right-side proofs are not symmetric. Recall that the original definition of `AbsFuncContig` applies `pushR` to a recursive application of `AbsFuncContig` on a smaller index range. The third lemma gives a similar rewriting in terms of `pushL`. In the full proof script, each of these lemmas is proved using an inductive schema, either over the constructors of the deque data type, or over the length of deque.

This concludes our overview of the linearizability proof for the array-based algorithm. It is straightforward to see that this algorithm is lock-free. Specifically, by examining the algorithm, we can see that an operation can take infinitely many steps without completing only if it executes infinitely many failing DCAS's. Furthermore, it is easy to see that each DCAS fails only if a shared variable that it accesses changes within one loop iteration. Such changes can occur only as the result of a successful DCAS. Furthermore, each time a processor executes a successful DCAS, it completes an operation before executing another DCAS. Thus, if one operation has infinitely many failing DCAS's, then infinitely many operations of other processors complete. This concludes the proof of lock-freedom for the array-based algorithm.

5.2 Proof of the pointer-based implementation

The proof of the pointer-based implementation is quite similar in structure to that of the array-based implementation, but presents some interesting difficulties. Pointer-based data structures are often difficult to mechanically verify, usually because of the potential for aliasing. The pointer-based deque data structure is fairly simple, because the pointers joining the nodes that represent the deque organize those nodes into a doubly-linked list. We express this fact as part of the representation invariant in order to allow the theorem prover to make use the information. We do this by introducing several *auxiliary variables*, variables that are used in the proof, but are not actually part of the program (i.e., they do not affect values assigned to implementation variables or affect control flow). These variables collectively define the set of nodes that currently form the representation of the deque, and their order in the linked lists. We introduce three auxiliary variables: `S`, a map from the (infinite set of) integers to node pointers, and integer indices `L` and `R`. The subsequence of `S` from `L` to `R`, inclusive, is the sequence of nodes in the doubly-linked list that forms the deque. Because these vari-

```

(DEFPRD (RepInv left left_del right right_del value l r s)
  (AND
    ;; We'll talk first about the sequence of nodes.
    (LBL RgtL (> r l))

    ;; The left and right elements are the sentinels.
    (LBL LeftSent
      (AND
        (EQ (select s l) SL)
        (EQ (select value SL) sentL)))
    (LBL RightSent
      (AND
        (EQ (select s r) SR)
        (EQ (select value SR) sentR)))

    ;; All elements of the sequence are distinct.
    (LBL DistinctNodes
      (FORALL (i j)
        (IMPLIES
          (AND (>= i l) (<= i r) (>= j l) (<= j r)
            ;; For ints i and j, below says i < j.
            (<= (+ i 1) j)
            )
          (NEQ (select s i) (select s j)))))

    ;; The right pointers point forward in the sequence.
    (LBL RightPointers
      (FORALL (i)
        (IMPLIES
          (AND (>= i l) (< i r))
          (EQ (select right (select s i)) (select s (+ i 1))))))

    ;; And the left pointers point backwards.
    (LBL LeftPointers
      (FORALL (i)
        (IMPLIES
          (AND (> i l) (<= i r))
          (EQ (select left (select s i)) (select s (- i 1))))))

    ;; ... more conjuncts in later figure ...

```

Figure 24: Definition of representation invariant for the linked-list-based implementation (first part).

ables are not actually part of the implementation, we are free to assume that they are updated atomically at DCAS linearization points, along with the actual implementation variables modified by the DCAS.

Figure 24 presents the first portion of the representation invariant for the pointer-based implementation. This predicate has eight free variables. The first five of these are implementation variables, and the last three are values of the auxiliary variables discussed above. The implementation variables are *field maps*. We use such maps to represent fields of dynamically allocated objects referenced by pointers. A field `f` of such an object type is represented as a map, indexed by references to objects containing the field. The `left` and `left_del` fields together represent the pointer and deleted bits of the left pointer of a deque node, and the `right` and `right_del` maps represent the right pointers. The `value` field is another map, representing values stored in the deque nodes. The `l`, `r`, and `s` arguments are the auxiliary variables.

The invariant is a somewhat large conjunction. Figure 24 shows some of these conjuncts, Figure 24 the remaining ones. First, a well-formedness condition requires that the right index into the abstract map `s` be strictly greater than the left index. This inequality is strict because deque implementations always have at least two nodes, the left and right sentinels; the next two conjuncts require that the nodes indexed by `l` and `r` are those sentinels, and the corresponding

nodes have the correct values.

The **DistinctNodes** conjunct is important, because it addresses aliasing issues that are often difficult in mechanical verification of pointer-based data structures. By informing the prover that all nodes of interest within **s** are distinct, we allow it to infer that updates to fields of one node in the deque do not alter values of fields of other nodes. The **RightPointers** and **LeftPointers** conjuncts express the relationship between the sequence and the doubly-linked pointer structure.

The last two conjuncts of Figure 24 claim that if the inward-pointing pointer field in a sentinel has its “deleted” bit set, then the node referenced by the “value” portion of the pointer contains the “null” value.

The remaining conjuncts are shown in Figure 25. The first two of these conjuncts claim that if the inward-pointing pointer field in a sentinel has its “deleted” bit set, then the node referenced by the “value” portion of the pointer contains the “null” value.

Next there are two conjuncts that state what can be inferred from the converse: when a node pointed to by a sentinel is observed to contain the “null” value. It is possible that the “deleted” bit of the sentinel’s pointer is set, but this is not necessarily true; it may be that the node is the only non-sentinel node, and the *other* sentinel’s pointer to it has its “deleted” bit set (in which case the current sentinel’s “deleted” bit cannot be set).

Finally, the invariant has four conjuncts stating that nodes in the sequence that are not sentinels, and are not deleted, contain “real” values (values other than “null” and the two sentinel values.) These conjuncts are necessary to allow the inference that observation of a real value implies non-emptiness of the deque. The figure shows only the first of these conjuncts; the remaining three are similar, varying according to which sentinels have deleted bits set.

The abstraction function and related auxiliary functions are very similar to those used in the array-based case, with one significant difference. The abstraction function used in the array-based proof took the representation of the array used in the implementation as an argument, along with its **L** and **R** indices. This array mapped integers to values. In the pointer-based case, the abstraction function takes as arguments the auxiliary map **S** and its associated **L** and **R** indices as arguments. This array maps integers to deque node references, not to values; therefore, the abstraction function also takes the value of the **value** field map as an argument, so that that node pointers can be “dereferenced” to yield the values stored in them. Figure 36 in Appendix B shows the abstraction function for the pointer-based implementation.

Below we define the linearization points of the operations, and specify updates to auxiliary variables associated with some of those points. As usual, we describe only the right-side operations; the left-side operations are symmetric.

We first consider the **popRight** operation, shown in Figure 11. It returns in three places, at lines 5, 11, and 18. A **popRight** operation that returns at line 5 is linearized at the point where the load in line 3 is executed. A **popRight** operation that returns at line 11 is linearized at successful execution of the DCAS at line 9. Finally, a **popRight** operation that returns at line 18 is linearized at successful execution of the DCAS at line 16.

Next we consider the **pushRight** operation. It returns at two places, at line 3 and line 18. For the purposes of the proof, we assume that there is always sufficient memory available, so that the allocation at line 2 always succeeds.⁸

⁸Assigning a linearization point to executions in which the allocation

```
;; ... continuation of conjunction defining RepInv ...

(LBL LeftDeletedNodeHasNullVal
  (IMPLIES
    (EQ (select left_del SR) 1)
    (EQ (select value (select left SR)) NullVal)))

(LBL RightDeletedNodeHasNullVal
  (IMPLIES
    (EQ (select right_del SL) 1)
    (EQ (select value (select right SL)) NullVal)))

(LBL NullLeftValImpliesDeletedLeftOrRight
  (IMPLIES
    (EQ (select value (select right SL)) NullVal)
    (OR (EQ (select right_del SL) 1)
        (AND (EQ (select left SR)
              (select right SL))
             (EQ (select left_del SR) 1)
             (NEQ (select right_del SL) 1)
             )))))

(LBL NullRightValImpliesDeletedRightOrLeft
  (IMPLIES
    (EQ (select value (select left SR)) NullVal)
    (OR (EQ (select left_del SR) 1)
        (AND (EQ (select right SL)
              (select left SR))
             (EQ (select right_del SL) 1)
             (NEQ (select left_del SR) 1)
             )))))

;; There are four cases, depending on deletion status.
(LBL NonDelNonSentNodesHaveRealValsA
  (FORALL (kk)
    (PATS (select value (select s kk)))
    (IMPLIES
      (AND
        (NEQ (select right_del SL) 1)
        (NEQ (select left_del SR) 1)
        (> kk 1) (< kk r))
      (IsValid (select value (select s kk))))))

(LBL NonDelNonSentNodesHaveRealValsB
  (FORALL (k)
    (PATS (select val (select s k)))
    (IMPLIES
      (AND
        (NEQ (select right_del LeftSentinel) 1)
        (EQ (select left_del RightSentinel) 1)
        (> k 1) (< k (- r 1)))
      (IsValid (select val (select s k))))))

(LBL NonDelNonSentNodesHaveRealValsC
  (FORALL (k)
    (PATS (select val (select s k)))
    (IMPLIES
      (AND
        (EQ (select right_del LeftSentinel) 1)
        (NEQ (select left_del RightSentinel) 1)
        (> k (+ 1 1)) (< k r))
      (IsValid (select val (select s k))))))

(LBL NonDelNonSentNodesHaveRealValsD
  (FORALL (k)
    (PATS (select val (select s k)))
    (IMPLIES
      (AND
        (EQ (select right_del LeftSentinel) 1)
        (EQ (select left_del RightSentinel) 1)
        (> k (+ 1 1)) (< k (- r 1)))
      (IsValid (select val (select s k))))))

))
```

Figure 25: Definition of representation invariant for the linked-list-based implementation (second part).

tion at line 2 fails would require a detailed knowledge of the storage allocator; further, determining that the operation is correctly linearized at this point would require a definition of what “full” means in the abstract model.

A `pushRight` operation that returns at line 18 is linearized at successful execution of the DCAS at line 16. In this case, we further assume that the following updates to auxiliary variables occur atomically with the DCAS, if and only if it is successful:

```
S[R] = newL;
S[R+1] = SR;
R = R + 1;
```

That is, the new node is inserted in the sequence just to the left of the right sentinel.

In considering the linearization points for the `popRight` and `pushRight` operations, we have already covered all statements in these operations that potentially modify shared variables (and therefore all statements that may change the abstract deque value represented by those variables). It remains to consider such statements of the `deleteRight` operation (which has no linearization points) because some of these statements modify shared variables, and therefore we must show that they preserve the representation invariant, and that these modifications do not change the abstract value of the deque. These modifications occur via the DCAS operations at lines 11 and 23. In each case, we assume that auxiliary variables are updated atomically with the the DCAS, if and only if it is successful. Specifically, at Line 11, we assume the following updates:

```
S[R - 1] = SR;
R = R - 1;
```

That is, the rightmost non-sentinel node is removed from the sequence. At line 23, the updates reset the sequence to contain only the two sentinels:

```
L = 0;
R = 1;
S[0] = SL;
S[1] = SR;
```

As in the array-based case, we define a set of verification conditions that together imply that the algorithm preserves the representation invariant, that each linearization point corresponds to an appropriate transition of the abstract deque, and that the return value determined at each of these linearization points is consistent with that transition. Below we present a representative sample of these verification conditions; as stated earlier, full proof scripts are available.

We begin by presenting the verification condition that shows that any state that satisfies the initial conditions for the algorithm also satisfies the representation invariant and represents an empty deque; this verification condition is shown in Figure 26.

In order to represent any state that satisfies the initial conditions, we constrain the various field maps that together represent the implementation state to have the specified values at the required indices, as determined by the algorithm’s initial conditions. This is achieved by using the store operator on arbitrary values of these field maps.

Recall that there are two distinct sentinel nodes; these are referred to by the pointer constants `SL` and `SR`. The initial conditions require that these two nodes point to each other; that these pointers are not marked as deleted; and that the nodes contain the special values `sentL` and `sentR`. Further, the auxiliary map `S`, together with indices `L` and `R`, initially specify the node sequence consisting of `SL` and `SR` in that order. Thus, the verification condition shown in Figure 26

says that any state that satisfies the initial conditions of the algorithm also satisfies the representation invariant and represents the empty deque.

```
(FORALL (AnyS AnyLeft AnyLeftDel AnyRight AnyRightDel AnyVal)
(IMPLIES
(NEQ SL SR)
(AND
(RepInv
(store AnyLeft SR SL)
(store (store AnyLeftDel SR 0) SL 0)
(store AnyRight SL SR)
(store (store AnyRightDel SL 0) SR 0)
(store (store AnyVal SL sentL)
SR sentR)
0 1 (store (store AnyS 0 SL) 1 SR))
(EQ
(AbsFunc
(store (store AnyS 0 SL) 1 SR)
0 1
(store (store AnyLeftDel SR 0) SL 0)
(store (store AnyRightDel SL 0) SR 0)
(store (store AnyVal SL sentL)
SR sentR)
EmptyQ)
)))
```

Figure 26: Verification condition for initial conditions of linked-list-based deque implementation.

We next present the verification condition for the DCAS linearization point at line 16 of the `pushRight` operation shown in Figure 13; this verification condition is shown in Figure 27. This DCAS is a linearization point only if it succeeds. The verification condition is an implication whose antecedent is true only of reachable states in which the DCAS would succeed. Specifically, the first three conjuncts of the antecedent say that the representation invariant holds, that the input value is a “real” value (i.e., is not one of the special values), and that left pointer of the right sentinel is not marked as deleted. To see why this last conjunct is justified, observe that the algorithm cannot reach line 16 if `oldL.deleted` is true (see line 7). The last conjunct of the antecedent captures the allocation and initialization of a new node. This is expressed using the `NewWRTSeq` predicate, shown in Figure 37, which expresses two important facts about the newly-allocated node. First, it says that the new node pointer is distinct from all node pointers in the current sequence. Second, it says that the field maps comprising the implementation state correctly reflect the values stored into the fields of the new node by the algorithm. Both of these assumptions are justified by the fact that the node is allocated at line 2, and no pointer to this node is stored in any shared variable before the successful execution of the DCAS at line 16. Thus, no other thread can either modify the contents of the node before this time, or cause this node to become part of the sequence.

The consequent of the verification condition in Figure 27 says that the representation invariant holds in the state produced by successfully executing the DCAS, and that the abstract deque in the new state is the result of applying the `pushR` operation to the abstract deque of the previous state. It remains to describe how the new state is expressed in terms of the old. The DCAS updates the left pointer of the right sentinel, and the right pointer of the node immediately to the left of the right sentinel (the DCAS confirms that the right pointer of this node points to the right sentinel). Thus, the `left` and `right` field maps in the post-DCAS state are updated accordingly. In addition, recall that the auxiliary

sequence is updated atomically with the DCAS execution as specified earlier, so that the new node is inserted into the sequence.

```
(FORALL (left left_del right right_del value l r s newR v)
(IMPLIES
(AND
(RepInv left left_del right right_del value l r s)
(IsVal v)
(NEQ (select left_del SR) 1) ;; Confirmed by DCAS
(NewWRTSeq newR l r s value v
left (select left SR)
right SR
left_del right_del)
)
(AND
(LBL RepInvPreserved
(RepInv (store left SR newR)
left_del
(store right (select left SR) newR)
right_del
value
l (+ r 1)
(store (store s (+ r 1) SR)
r newR)))
(LBL ProperTransition
(EQ (AbsFunc
(store (store s (+ r 1) SR) r newR)
l (+ r 1)
left_del right_del value)
(pushR (AbsFunc s l r left_del right_del value) v))
)
)))
```

Figure 27: Verification condition for line 16 of linked-list `pushRight`, Figure 13.

Next we consider one of the verification conditions for the `popRight` operation. This operation has two linearization points. The verification condition for the DCAS at Line 16 is quite similar to the one just discussed. We therefore present the verification condition for the linearization point at line 3, which is shown in Figure 28. This predicate says that, in a state in which the representation invariant holds and the value field of the node pointed to by the left pointer of the right sentinel is the “sentL” value, the abstract deque state is the empty deque. There is a subtlety here: the right sentinel’s left pointer is read in line 3 (the linearization point), and the value field of the node it references is read separately, at line 4. The operation returns at line 5 only if that value is the distinguished “sentL” value. Our proof obligation is that the abstract deque value *at the linearization point* is the empty deque. If the right sentinel’s left pointer references the left sentinel, then the representation invariant and abstraction function together imply that the implementation state represents the empty deque. We infer that the pointer value read at line 3 references the left sentinel by observing the “sentL” value in the node’s value field at line 4. However, because this read occurs *after* the linearization point, one could object that the node’s value field may not have contained “sentL” at the linearization point. We answer this objection by noting that the representation invariant implies that the fixed sentinel nodes always contain the proper special values, and that no other node ever does.

The last verification condition we present is for the successful execution of the DCAS at line 11 of the `deleteRight` operation; this predicate is shown in Figure 29 and Figure 15 shows this transition pictorially. This DCAS is not a linearization point, but it does represent a transition that

```
;; popR -- read at line 3; return at line 5
(FORALL (left left_del right right_del value l r s)
(IMPLIES
(AND
(RepInv left left_del right right_del value l r s)
(EQ (select value (select left SR)) sentL)
)
(EQ (AbsFunc s l r left_del right_del value) EmptyQ)))
```

Figure 28: Verification condition for line 3 of linked-list `popRight`, Figure 11.

modifies shared and auxiliary variables. Therefore, we must show that it preserves the representation invariant and does not change the abstract value of the deque. As for the verification conditions discussed above, we first explain why the antecedent captures the conditions under which the DCAS can succeed, and then explain how the consequent embodies the proof obligations just mentioned.

As usual, the first conjunct of the antecedent says that the representation invariant holds. The remaining two conjuncts are facts about the memory state that are confirmed by successful execution of the DCAS: that the left pointer of the right sentinel is marked as deleted, and that the right pointer of the node two places the left of the right sentinel is equal to the left pointer of the right sentinel.

The consequent of the verification condition says that the representation invariant holds in the post-DCAS state, and that the abstract value of the deque is the same in the post-DCAS state as it was in the pre-DCAS state. The successful DCAS changes the left pointer of the right sentinel into a non-marked pointer to the node two places to the left of the right sentinel (in the pre-DCAS state), and changes the right pointer of this node to be a non-marked pointer to the right sentinel. In other words, the DCAS “splices out” the first non-sentinel node from the right, as shown in Figure 15. Some readers may be concerned that *four* variables of our model of the implementation state are modified by the *double* compare-and-swap operation. However, recall that each pointer value and its associated “deleted” bit are stored together in a single memory word that can be accessed by the DCAS.

In addition, recall that the auxiliary variables representing the node sequence are modified atomically at the same time the DCAS is executed, removing the deleted node from the sequence.

This concludes our overview of the linearizability proof for the linked-list-based algorithm. The lock-freedom argument for this algorithm is slightly more difficult than the one for the array-based algorithm. The reason is that it is not the case for this algorithm that each time a processor executes a successful DCAS, it completes an operation before executing another DCAS. In particular, it is possible for a processor to execute infinitely many successful DCAS’s without ever completing an operation because the DCAS’s in the `deleteRight` and `deleteLeft` procedures can succeed without the processor that executed them ever completing an operation. We therefore need a slightly more involved argument for this algorithm, as follows.

First, observe that a successful DCAS in the `popRight` or `pushRight` operations (or their symmetric counterparts) results in a completed operation. Therefore, assume that in some execution history H , some processor p takes infinitely many steps without completing an operation, and that only finitely many of the DCAS’s in `popRight`, `pushRight`, and

```

(FORALL (left left_del right right_del value l r s)
  (IMPLIES
    (AND
      (RepInv left left_del right right_del value l r s)
      (EQ (select left_del SR) 1)
      (EQ (select right (select left (select left SR)))
          (select left SR)))
    )
    (AND
      (LBL RepInvPreserved
        (RepInv
          (store left SR
            (select left (select left SR)))
          (store left_del SR 0)
          (store right (select left (select left SR))
            SR)
          (store right_del
            (select left (select left SR)) 0)
          value
            1
            (- r 1)
            (store s (- r 1) SR))
        )
      (LBL AbsValPreserved
        (EQ (AbsFunc
          (store s (- r 1) SR)
          1
          (- r 1)
          (store left_del SR 0)
          (store right_del
            (select left (select left SR)) 0)
          value)
          (AbsFunc s l r left_del right_del value)))
        )
      )
    )))

```

Figure 29: Verification condition for line 11 of linked-list `deleteRight`, Figure 17.

their symmetric counterparts succeed. We consider two cases.

In the first case, H contains only finitely many successful DCAS's. We derive a contradiction in this case. First, it is easy to see that each attempted DCAS fails only if some other DCAS succeeds during the loop iteration of the failed DCAS (because each DCAS supplies values for the “old” arguments that it has read during that iteration). Thus, if there are infinitely many *attempted* DCAS's, then there are infinitely many *successful* ones (assuming a finite number of processors). Therefore, because there are only finitely many successful DCAS's, after some point, there are no more attempted DCAS's by any processor and processor p takes infinitely many steps after this point. Because there are no more successful DCAS's, the shared variables do not change, so the shared state remains the same while p takes these steps. We use the representation invariant to reason that this is impossible.

The remainder of the argument for this case is symmetric for the left and right sides; we present the argument only for the right side. First, by examining the code for `popRight` and `pushRight`, we can see that during every loop iteration of any operation, p either calls `deleteRight` or attempts a DCAS. Therefore, p executes the loop in `deleteRight` infinitely often (either by virtue of invoking `deleteRight` infinitely often, or by executing the loop infinitely often in some invocation of `deleteRight`).

The `deleteRight` procedure can return without attempting a DCAS only if it observes `SR->L.deleted` to be false. In this case, p will invoke `deleteRight` again only if it subsequently observes `SR->L.deleted` to be true, which implies that some DCAS succeeded, a contradiction. Thus p executes the loop in `deleteRight` infinitely often without re-

turning and without attempting a DCAS. This implies that each time around the loop, either the test at line 8 or the test at line 18 fails. To see that neither scenario is possible, we appeal to the representation invariant (recall that the shared state does not change during the loop executions under consideration). If the test at line 8 fails, then the loads at lines 5 and 7 show that the right pointer of the left neighbor of the node pointed to by `oldL.ptr` does not point to that node; this contradicts the representation invariant, which says (in the conjuncts labeled `RightPointers` and `LeftPointers`) that the nodes in the deque always form a doubly-linked list. If the test at line 18 fails, then loads at lines 3, 5, 6, and 17 show a state in which the second node from the right contains a null value, but that the right pointer of the left sentinel is not marked as deleted. This again contradicts the representation invariant, which says (in the last four conjuncts of figure 25) that a node can contain a null value only if it is adjacent to a sentinel that has its inward pointer marked as deleted.

In the second case, H contains infinitely many successful DCAS's, but only finitely many from the `popRight` and `pushRight` operations (and their symmetric counterparts). Therefore, after some point, there are infinitely many successful DCAS's, all of which are in `deleteRight` or `deleteLeft`. By examining the code, we can see that each successful DCAS in these operations changes a `deleted` bit from *true* to *false*. This can happen infinitely often only if these bits are changed from *false* to *true* infinitely often, which contradicts the assumption that after some point only DCAS's in the `deleteRight` and `deleteLeft` operations succeed (because only the DCAS's in the pop operations change `deleted` bits to *true*).

This completes the lock-freedom proof for the linked-list-based algorithm.

6 Concluding Remarks

We have presented two new DCAS-based lock-free concurrent deque implementations. One is array-based, and the other uses a linked list representation. Both support non-interfering concurrent access to opposite ends of the deque whenever possible.

As discussed in Section 1.1, this work was the beginning of a line of research involving lock-free implementations of concurrent data structures based on DCAS. This experience clearly demonstrates that synchronization primitives that can access more than one shared memory location atomically allow us to design lock-free algorithms that are substantially simpler than the best-known counterparts that use only single-location synchronization primitives. Furthermore, these latter implementations are complicated and entail significant overhead; it seems very likely that our DCAS-based algorithms would perform much better. (Of course, without detailed knowledge of the implementation of a particular system supporting DCAS, we cannot quantify this comparison.) Thus our work adds weight to the argument that stronger synchronization primitives are needed for scalable synchronization. However, our work also has implications for existing systems that do not support such primitives. In particular, insights gained from this line of work have already led us to ideas for practical algorithms that do not depend on them.

We should also caution that, although the use of DCAS did facilitate relatively simple algorithms with desirable properties, the algorithms are still far from trivial. Therefore, we

do not believe that DCAS is a “silver bullet” for easy solutions to all difficult synchronization problems.

7 Acknowledgments

We would like to thank Steve Heller, Maurice Herlihy, Doug Lea, and the anonymous referees for their many suggestions and comments.

Sun, Sun Microsystems, the Sun logo, Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

References

- [1] AFEK, Y., MERRITT, M., TAUBENFELD, G., AND TOUITOU, D. Disentangling multi-object operations. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing* (August 1997), pp. 111–120. Santa Barbara, CA.
- [2] AGESEN, O., AND CARTWRIGHT JR., R. S. Platform independent double compare and swap operation, Dec. 1998. U.S. Patent Application Express Mail Number EL092132504US Ref P3368.
- [3] AGESEN, O., DETLEFS, D., FLOOD, C., GARTHWAITE, A., MARTIN, P., SHAVIT, N., AND STEELE, G. DCAS-based concurrent dequeues. In *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures* (2000), pp. 137–146.
- [4] ARORA, N. S., BLUMOFFE, B., AND PLAXTON, C. G. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures* (1998).
- [5] ATTIYA, H., AND DAGAN, E. Universal operations: Unary versus binary. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing* (May 23–26 1996). Phila. PA.
- [6] ATTIYA, H., LYNCH, N., AND SHAVIT, N. Are wait-free algorithms fast? *Journal of the ACM* 41, 4 (July 1994), 725–763.
- [7] ATTIYA, H., AND RACHMAN, O. Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal on Computing* 27, 2 (Mar. 1998), 319–340.
- [8] BARNES, G. A method for implementing lock-free shared data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures* (June 1993), pp. 261–270.
- [9] BERSHAD, B. N. Practical considerations for non-blocking concurrent objects. In *Proceedings 13th IEEE International Conference on Distributed Computing Systems* (May 25–28 1993), IEEE Computer Society Press, pp. 264–273. Los Alamitos CA.
- [10] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*, 1st ed. McGraw Hill, 1989.
- [11] DETLEFS, D., FLOOD, C., GARTHWAITE, A., MARTIN, P., SHAVIT, N., AND STEELE, G. Even better DCAS-based concurrent dequeues. In *Proceedings of the 14th International Conference on Distributed Computing* (2000), pp. 59–73.
- [12] DETLEFS, D., MARTIN, P., MOIR, M., AND STEELE, G. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing* (2001), pp. 190–199.
- [13] DETLEFS, D. L., LEINO, K. R. M., SAXE, J. B., AND NELSON, G. Extended static checking. Tech. Rep. SRC Research Report 159, Compaq Systems Research Center, December 1998.
- [14] DETLEFS, D. L., NELSON, G., AND SAXE, J. B. <http://www.research.compaq.com/SRC/esc/Simplify.html>. Home page for the Simplify theorem prover.
- [15] GOSLING, J., JOY, B., AND STEELE JR., G. L. *The Java™ Language Specification*. Addison-Wesley, 2550 Garcia Avenue, Mountain View, CA 94043-1100, 1996.
- [16] GREENWALD, M. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, 8 1999.
- [17] GREENWALD, M. B., AND CHERITON, D. R. The synergy between non-blocking synchronization and operating system structure. In *2nd Symposium on Operating Systems Design and Implementation* (October 28–31 1996), pp. 123–136. Seattle, WA.
- [18] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann Publishers, 1995.
- [19] HERLIHY, M. Wait-free synchronization. *ACM Transactions On Programming Languages and Systems* 13, 1 (Jan. 1991), 123–149.
- [20] HERLIHY, M. A methodology for implementing highly concurrent data structures. *ACM Transactions on Programming Languages and Systems* 15, 5 (Nov. 1993), 745–770.
- [21] HERLIHY, M. P., AND MOSS, J. Transactional memory: Architectural support for lock-free data structures. Tech. Rep. CRL 92/07, Digital Equipment Corporation, Cambridge Research Lab, 1992.
- [22] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions On Programming Languages and Systems* 12, 3 (July 1990), 463–492.
- [23] KNUTH, D. E. *The Art of Computer Programming: Fundamental Algorithms*, 2nd ed. Addison-Wesley, 1968.
- [24] MARTIN, P., MOIR, M., AND STEELE, G. Dcas-based concurrent dequeues supporting bulk allocation. Submitted to Computing: The Australasian Theory Symposium, August 2001.
- [25] MASSALIN, H., AND PU, C. A lock-free multiprocessor OS kernel. Tech. Rep. TR CUCS-005-9, Columbia University, New York, NY, 1991.

- [26] MICHAEL, M. M., AND SCOTT, M. L. Correction of a memory management method for lock-free data structures. Technical Report TR 599, Computer Science Department, University of Rochester, 1995.
- [27] MOTOROLA. *MC68020 32-Bit Microprocessor User's Manual*, 2nd ed. Prentice-Hall, 1986.
- [28] MOTOROLA. *MC68030 User's Manual*. Prentice-Hall, 1989.
- [29] RINARD, M. C. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems* 17, 4 (Nov. 1999), 337–371.
- [30] SHAVIT, N., AND TOUITOU, D. Software transactional memory. *Distributed Computing* 10, 2 (February 1997), 99–116.
- [31] STEELE JR., G. L. *Common Lisp*. Digital Press, Digital Equipment Corporation, 1990.

A Appendix: The matching left hand side code

```

0 val popLeft {
1   newS = "null";
2   while (true) {
3     oldL = L;
4     newL = (oldL + 1) mod length_S;
5     oldS = S[newL];
6     if (oldS == "null") {
7       if (oldL == L)
8         if (DCAS(&L, &S[newL],
9                 oldL, oldS, oldL, oldS))
10          return "empty";
11    }
12    else {
13      saveL = oldL;
14      if (DCAS(&L, &S[newL],
15              &oldL, &oldS, newL, newS))
16        return oldS;
17      else if (oldL == saveL) {
18        if (oldS == "null") return "empty";
19      }
20    }
21  }
22 }

```

Figure 30: The array-based deque left-hand-side pop.

```

1 val pushLeft(val v) {
2   while (true) {
3     oldL = L;
4     newL = (oldL - 1) mod length_S;
5     oldS = S[oldL];
6     if (oldS != "null") {
7       if (oldL == L)
8         if (DCAS(&L, &S[oldL],
9                 oldL, oldS, oldL, oldS))
10          return "full" ;
11    }
12    else {
13      saveL = oldL;
14      if (DCAS(&L, &S[oldL],
15              &oldL, &oldS, newL, v))
16        return "okay";
17      else if (oldL == saveL)
18        return "full";
19    }
20  }
21 }

```

Figure 31: The array-based deque left-hand-side push.

```

1 val popLeft() {
2   while (true) {
3     oldR = SL->R;
4     v = oldL.ptr->value;
5     if (v == "sentR") return "empty";
6     if (oldR.deleted == true)
7       deleteLeft();
8     else if (v == "null") {
9       if (DCAS(&SL->R, &oldR.ptr->value,
10              oldR, v, oldR, v))
11        return "empty";
12    }
13    else {
14      newR.ptr = oldR.ptr;
15      newR.deleted = true;
16      if (DCAS(&SL->R, &oldR.ptr->value,
17              oldR, v, newR, "null"))
18        return v;
19    }
20  }
21 }

```

Figure 32: The linked-list-based deque - left side pop.

```

1 val pushLeft(val v) {
2   newR.ptr = new Node();
3   if (newR.ptr == "null") return "full";
4   newR.deleted = false;
5   while (true) {
6     oldR = SL->R;
7     if (oldR.deleted == true)
8       deleteLeft();
9     else {
10      newR.ptr->L.ptr = SR;
11      newR.ptr->L.deleted = false;
12      newR.ptr->R = oldR;
13      newR->value = v;
14      oldRL.ptr = SL;
15      oldRL.deleted = false;
16      if (DCAS(&SL->R, &oldR.ptr->L,
17              oldR, oldRL, newR, newR))
18        return "okay";
19    }
20  }
21 }

```

Figure 33: The linked-list-based deque - left side push.

```

1 deleteLeft() {
2   while (true) {
3     oldR = SL->R;
4     if (oldR.deleted == false) return;
5     oldRR = oldR.ptr->R.ptr;
6     if (oldRR->value != "null") {
7       oldRRL = oldRR->L;
8       if (oldR.ptr == oldRRL.ptr) {
9         newL.ptr = SL;
10        newL.deleted = false;
11        if (DCAS(&SL->R, &oldRR->L,
12              oldR, oldRRL, oldRR, newL))
13          return;
14      }
15    }
16    else { /* there are two null items */
17      oldL = SR->L;
18      newR.ptr = SR;
19      newR.deleted = false;
20      newL.ptr = SL;
21      newL.deleted = false;
22      if (oldL.deleted)
23        if (DCAS(&SL->R, &SR->L,
24              oldR, oldL, newR, newL))
25          return;
26    }
27  }
28 }

```

Figure 34: The linked-list-based deque - left side delete.

B Appendix: More details of the proofs

In this appendix, we present details left out of the expositions of the proofs.

```

;; Applications of different constructors are distinct.
(FORALL (v) (NEQ (singleton v) EmptyQ))
(FORALL (q1 q2)
  (PATS (NEQ (concat q1 q2) EmptyQ))
  (IMPLIES (NEQ q1 EmptyQ) (NEQ (concat q1 q2) EmptyQ)))
(FORALL (q1 q2)
  (PATS (NEQ (concat q1 q2) EmptyQ))
  (IMPLIES (NEQ q2 EmptyQ) (NEQ (concat q1 q2) EmptyQ)))

;; EmptyQ is a left and right identity for concat.
(FORALL (q) (EQ (concat q EmptyQ) q))
(FORALL (q) (EQ (concat EmptyQ q) q))

;; concat is associative.
(FORALL (q1 q2 q3) (EQ (concat q1 (concat q2 q3))
  (concat (concat q1 q2) q3)))

;; Define pushL and pushR
(FORALL (q v)
  (PATS (pushL q v))
  (EQ (pushL q v) (concat (singleton v) q)))
(FORALL (q v)
  (PATS (pushR q v))
  (EQ (pushR q v) (concat q (singleton v))))

;; Now define the peek observers, peekR and peekL.
;; Neither is defined on empty deques.
(FORALL (v)
  (PATS (peekR (singleton v)))
  (EQ (peekR (singleton v)) v))
(FORALL (q1 q2)
  (PATS (peekR (concat q1 q2)))
  (IMPLIES (NEQ q2 EmptyQ)
    (EQ (peekR (concat q1 q2)) (peekR q2))))

(FORALL (v)
  (PATS (peekL (singleton v)))
  (EQ (peekL (singleton v)) v))
(FORALL (q1 q2)
  (PATS (peekL (concat q1 q2)))
  (IMPLIES (NEQ q1 EmptyQ)
    (EQ (peekL (concat q1 q2)) (peekL q1))))

;; Now we define "pop" mutators.
(FORALL (v)
  (PATS (popR (singleton v)))
  (EQ (popR (singleton v)) EmptyQ))
(FORALL (q1 q2)
  (PATS (popR (concat q1 q2)))
  (IMPLIES (NEQ q2 EmptyQ)
    (EQ (popR (concat q1 q2)) (concat q1 (popR q2))))))

(FORALL (v)
  (PATS (popL (singleton v)))
  (EQ (popL (singleton v)) EmptyQ))
(FORALL (q1 q2)
  (PATS (popL (concat q1 q2)))
  (IMPLIES (NEQ q1 EmptyQ)
    (EQ (popL (concat q1 q2)) (concat (popL q1) q2))))

;; We define a len function, for use in induction.
(EQ (len EmptyQ) 0)
(FORALL (v) (EQ (len (singleton v)) 1))
(FORALL (q1 q2)
  (PATS (len (concat q1 q2)))
  (EQ (len (concat q1 q2)) (+ (len q1) (len q2))))

```

Figure 35: Axiomatization of abstract deque values.

```

;; Define AbsFuncContig.
;; (This version takes 'value', to translate nodes to values.)
(FORALL (s i j value)
  (IMPLIES (<= j i)
    (EQ (AbsFuncContig s i j value) EmptyQ)))
;; We introduce AbsFuncContigRest here to control the matching.
(FORALL (i j s value)
  (PATS (AbsFuncContig s i j value))
  (IMPLIES (< i j)
    (EQ (AbsFuncContig s i j value)
      (pushR (AbsFuncContigRest s i (- j 1) value)
        (select value (select s (- j 1)))))))
;; ...but it's really just another name for the same function.
(FORALL (i j s value)
  (PATS (AbsFuncContig s i j value))
  (EQ (AbsFuncContig s i j value) (AbsFuncContigRest s i j value)))

(FORALL (s i value)
  (EQ (AbsFuncContigRest s i i value) EmptyQ))

;; Now define AbsFunc in terms of AbsFuncContig.
;; There are 4 definitions, depending on the deleted bits.
(FORALL (s l r left_del right_del value)
  (PATS (AbsFunc s l r left_del right_del value))
  (IMPLIES
    (AND (NEQ (select right_del SL) 1)
      (NEQ (select left_del SR) 1))
    (EQ (AbsFunc s l r left_del right_del value)
      (AbsFuncContig s (+ 1 1) r value))))

(FORALL (s l r left_del right_del value)
  (PATS (AbsFunc s l r left_del right_del value))
  (IMPLIES
    (AND (NEQ (select right_del SL) 1)
      (EQ (select left_del SR) 1))
    (EQ (AbsFunc s l r left_del right_del value)
      (AbsFuncContig s (+ 1 1) (- r 1) value))))

(FORALL (s l r left_del right_del value)
  (PATS (AbsFunc s l r left_del right_del value))
  (IMPLIES
    (AND (EQ (select right_del SL) 1)
      (NEQ (select left_del SR) 1))
    (EQ (AbsFunc s l r left_del right_del value)
      (AbsFuncContig s (+ 1 2) r value))))

(FORALL (s l r left_del right_del value)
  (PATS (AbsFunc s l r left_del right_del value))
  (IMPLIES
    (AND (EQ (select right_del SL) 1)
      (EQ (select left_del SR) 1))
    (EQ (AbsFunc s l r left_del right_del value)
      (AbsFuncContig s (+ 1 2) (- r 1) value))))

(DEFUN NewWRTSeq newNode l r s
  value v
  left leftPtr
  right rightPtr
  left_del right_del)
(AND
  (FORALL (i)
    (IMPLIES (AND (>= i 1) (<= i r))
      (NEQ (select s i) newNode)))
  (EQ (select value newNode) v)
  (EQ (select left newNode) leftPtr)
  (EQ (select right newNode) rightPtr)
  (NEQ (select left_del newNode) 1)
  (NEQ (select right_del newNode) 1)
  ))

```

Figure 36: Abstraction function for linked-list-based implementation.

Figure 37: Definition of *NewWRTSeq*.