Dartmouth College

Dartmouth Digital Commons

Dartmouth College Ph.D Dissertations | Theses and Dissertations

11-15-2003

On the Complexity of Implementing Certain Classes of Shared Objects

King Yang Tan
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/dissertations

Part of the Computer Sciences Commons

Recommended Citation

Tan, King Yang, "On the Complexity of Implementing Certain Classes of Shared Objects" (2003). *Dartmouth College Ph.D Dissertations*. 69.
https://digitalcommons.dartmouth.edu/dissertations/69

This Thesis (Ph.D.) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Ph.D Dissertations by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.

Dartmouth College

Computer Science

Technical Report 2003-475

On the Complexity of Implementing Certain Classes of Shared Objects

A Thesis
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of
Doctor of Philosophy
in
Computer Science
by
King Yang Tan
DARTMOUTH COLLEGE
Hanover, New Hampshire
November 15, 2003

Examining Committee:

Prasad Jayanti (Chair)

Scot Drysdale

Vassos Hadzilacos
(University of Toronto)

Sean Smith

Carol Folt
Dean of Graduate Studies

Abstract

We consider shared memory systems in which asynchronous processes cooperate with each other by communicating via shared data objects, such as counters, queues, stacks, and priority queues. The common approach to implementing such shared objects is based on locking: To perform an operation on a shared object, a process obtains a lock, accesses the object, and then releases the lock. Locking, however, has several drawbacks, including convoying, priority inversion, and deadlocks. Furthermore, lock-based implementations are not fault-tolerant: if a process crashes while holding a lock, other processes can end up waiting forever for the lock.

Wait-free linearizable implementations were conceived to overcome most of the above drawbacks of locking. A wait-free implementation guarantees that if a process repeatedly takes steps, then its operation on the implemented data object will eventually complete, *regardless* of whether other processes are slow, or fast, or have crashed.

In this thesis, we first present an efficient wait-free linearizable implementation of a class of object types, called *closed* and *closable* types, and then prove time and space lower bounds on wait-free linearizable implementations of another class of object types, called *perturbable* types.

- We present a wait-free linearizable implementation of n-process closed and closable types (such as *swap*, *fetch&add*, *fetch&multiply*, and *fetch&*Φ, where Φ is any of the boolean operations *and*, *or*, or *complement*) using registers that support *load-link* (LL) and *store-conditional* (SC) as base objects.

  The time complexity of the implementation grows linearly with contention, but is never more than $O(\log^2 n)$. We believe that this is the first implementation of a class of types (as opposed to a specific type) to achieve a sub-linear time complexity.

- We prove linear time and space lower bounds on the wait-free linearizable implementations of n-process perturbable types (such as *increment*, *fetch&add*, modulo k *counter*, LL/SC bit, k-valued *compare&swap* (for any $k \geq n$), single-writer *snapshot*) that use resettable consensus and historyless objects (such as registers that support *read* and *write*) as base objects.

  This improves on some previously known $\Omega(\sqrt{n})$ space complexity lower bounds. It also shows the near space optimality of some known wait-free linearizable implementations.

Preface

I will always fondly remember my Dartmouth years as years of intellectual growth and spiritual exploration. For the privilege of engaging in such intellectual and spiritual journeys, I am grateful to Dartmouth College, and its Department of Computer Science.

Professor Prasad Jayanti epitomizes the ideal scholar and teacher, with empathy and compassion towards all. His creativity, intelligence, professional and personal integrity have served, and will continue to serve, as enduring guideposts in my life's journey. He is also a past master of the art of lucid and elegant exposition, both in prose and with the spoken word. In the classroom, he can be relied upon to inspire in his students a zeal to excel. In all these respects, I have benefited greatly from his personal example, scholarly guidance, and counsel.

He is truly a co-author of this thesis, the source of whatever merit it possesses. However, all errors and imperfections are my responsibility.

Professor Jayanti's impact on me has not been confined within the bounds of Computer Science. On a chilly autumn evening in November 2001, I bought two collections of Krishnamurti's writings at the open air Riverside Walk Market, under the Waterloo Bridge on the South Bank in London. I would not have done so without the influence of Professor Jayanti.

This thesis is based on previously published papers [JTT96, JTT00, CJT98], co-authored with Prasad Jayanti, Sam Toueg, and Tushar Deepak Chandra. None of this thesis would have been possible without their invaluable contribution.

Professor Sam Toueg of the University of Toronto supervised my research that resulted in Part Two of this thesis. I would like to express my gratitude for his guidance during my days at Cornell University.

Tushar Deepak Chandra of IBM Research is the inventor of an algorithm that eventually evolved into the algorithms in Part One of this thesis. He deserves a great deal of credit for this reason, besides many others.

Fruitful discussions, mostly on distributed computing, with my fellow students Jason Liu, Srdjan Petrovic, Michael Fromberger, Feng Cao, Gregory Friedland, Amir Katz, Ajinkya Mukhopadhyay, have enhanced my appreciation of distributed algorithms and resulted in some publications. I would like to express my sincere thanks to each one of them, and to others whom I have neglected, unintentionally, to mention.

Professors Scot Drysdale and Sean Smith of Dartmouth College, and Professor Vassos Hadzilacos of the University of Toronto, served on my thesis committee, and provided valuable advice that improved the quality of this thesis. It has been my honor to receive the benefit of their expert scholarly opinions. I am indeed grateful to them.

Ashvin Dsouza generously invited me into his personal, family, and professional life, and his home, since our days together at Cornell. I greatly treasure his friendship, generosity of spirit, and kindness of heart. I truly cherish the love, affection, and warmth that I have always received from Ashvin, Maeve, Kiran, and Rohit.

The hospitality and generosity extended to me by the Jayanti family — Prasad, Santha, Sucharita, and Siddharta — will forever be a cherished memory of my sojourn at Hanover, New Hampshire.

Finally, I can never adequately discharge my debt of gratitude to my mother, sisters, brothers, sisters-in-law, brothers-in-law, nieces, and nephews. Their love and support have made my life's journey possible in innumerable ways.

I had three pieces of limestone on my desk,
but I was terrified to find that they required to be dusted daily,
when the furniture of my mind was all undusted still,
and threw them out the window in disgust.

Henry David Thoreau
Walden

But often, in the world's most crowded streets,
But often, in the din of strife,
There rises an unspeakable desire
After the knowledge of our buried life;
A thirst to spend our fire and restless force
In tracking out our true, original course;
A longing to inquire
Into the mystery of this heart which beats
So wild, so deep in us - to know
Whence our lives come and where they go.

Matthew Arnold
The Buried Life

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.

T.S. Eliot
Little Gidding

Contents

Preface **ii**

1 Introduction **1**

1.1 Background . . . 1

1.1.1 Shared Memory System . . . 1

1.1.2 Shared Objects and their Implementation . . . 2

1.1.3 Lock-based, Wait-free, and Nonblocking Implementation . . . 2

1.1.4 Linearizability . . . 3

1.1.5 Deterministic and Randomized Implementation . . . 3

1.2 Contributions of the Thesis . . . 3

1.2.1 A Polylog Time Wait-Free Construction for Closed Objects . . . 4

1.2.1.1 Our Contribution . . . 4

1.2.1.2 Hardware support . . . 5

1.2.2 Time and Space Lower Bounds for Nonblocking Implementations . . . 5

1.3 Organization of the Thesis . . . 7

2 Model **8**

2.1 Object Type . . . 8

2.2 Common Object Types . . . 8

2.2.1 Test&Set . . . 8

2.2.2 Swap . . . 8

2.2.3 Compare&Swap . . . 8

2.2.4 Fetch&Add . . . 9

2.2.5 Increment . . . 9

2.2.6 Fetch&Multiply . . . 9

2.2.7 Fetch&Φ . . . 9

2.2.8 LL, SC, and VL . . . 9

2.2.9 Resettable Consensus . . . 10

2.3 Closed and Closable Types . . . 11

2.4 Historyless Types . . . 13

2.5 Implementation . . . 14

2.6 Linearizability . . . 15

2.7 Wait-freedom . . . 16

2.8 Solo-termination . . . 16
2.9 Notation . . . 16
2.10 Shared-access and Local Time Complexity . . . 17
2.11 "Just Completes" and Solo-Termination Time Complexity . . . 17

I Wait-Free Construction for Closed Objects **18**

3 Unbounded Construction for Closed Objects **19**
3.1 Introduction . . . 19
3.2 Informal Description of the Construction . . . 19
3.2.1 Binary Tree Preliminaries . . . 19
3.2.2 How Operations are Represented . . . 20
3.2.3 Linearization Order . . . 21
3.2.4 Growing Trees Bottom-up . . . 22
3.2.5 How a Process Makes Progress . . . 22
3.3 The Data Structure . . . 25
3.3.1 The Representation of a List . . . 26
3.3.1.1 The Fields of a Cell . . . 26
3.3.1.2 The *Head* Variables . . . 27
3.3.2 Stability and Order Properties . . . 27
3.3.2.1 Stability and Order Properties for a Non-root Interior Cell . . . 27
3.3.2.2 Stability and Order Properties for a Leaf Cell . . . 28
3.3.2.3 Stability and Order Properties for a Root Cell . . . 29
3.3.3 Facts about the *Head* Variables . . . 29
3.3.4 Definition of Correct State . . . 29
3.4 How the Algorithm Works . . . 30
3.4.1 How `apply` Works . . . 30
3.4.2 How `promote` Works on a Non-root List . . . 35
3.4.3 How `promote` Works on a Root List . . . 35
3.4.4 How `append` Works . . . 36
3.4.4.1 Why Property P5 Holds . . . 36
3.4.4.2 Why Property P6 Holds . . . 37
3.4.5 How `percolateState` Works . . . 37
3.5 Proof of Correctness . . . 38
3.5.1 Proof of Progress . . . 38
3.5.1.1 Reachable Cell . . . 38
3.5.1.2 Head . . . 40
3.5.1.3 Ancestor and Descendant . . . 42
3.5.1.4 Precedence Relations . . . 42
3.5.1.5 Uniqueness of Parent . . . 44
3.5.1.6 Properties of `append` . . . 45
3.5.1.7 Properties of `promote` . . . 47
3.5.1.8 Properties of `apply` . . . 48

3.5.2 Proof of Linearizability . . . 50
3.5.2.1 Functions *invoca* and *leaf* . . . 51
3.5.2.2 Leaf Sequence . . . 51
3.5.2.3 Invocation Sequence . . . 52
3.5.2.4 Operation Sequence . . . 53
3.5.2.5 *Op* Field of a Cell . . . 53
3.5.2.6 *State* Field of a Cell . . . 54
3.5.2.7 Linearizability . . . 57
3.5.3 Summary . . . 57

4 Bounded Construction for Closed Objects **59**
4.1 General Principles . . . 59
4.2 Bounded Space Complexity* (BSC*) Implementation . . . 61
4.2.1 `release` procedure . . . 61
4.3 Proof of Correctness of BSC* . . . 62
4.4 Properties of BSC* . . . 73
4.4.1 Invalid Cells . . . 73
4.4.2 Properties of $c \rightarrow Free$ . . . 74
4.4.3 Properties of $c \rightarrow Retired$ . . . 77
4.4.4 Preliminary Lemmas . . . 78
4.4.5 Reading from Invalid Cells . . . 80
4.4.6 Writing to Invalid Cells . . . 84
4.5 Bounded Space Complexity (BSC) Implementation . . . 86
4.5.1 `selectCell` and `replace` procedures . . . 88
4.6 Proof of Correctness of BSC . . . 89
4.6.1 Provisional Correctness Proof . . . 89
4.6.2 Bounding the Number of Valid Cells . . . 90
4.6.3 Properties of `selectCell` . . . 91
4.6.4 Correctness of the BSC Implementation . . . 93

5 Contention-Sensitive Implementation **94**
5.1 General Principles . . . 94
5.2 The Implementation . . . 102
5.3 Proof of Correctness . . . 102
5.3.1 Proof of Token Scheme . . . 103
5.3.2 Bounding the Number of Valid Cells . . . 107
5.3.3 Proof of Correctness . . . 109

II Lower Bounds for Nonblocking Implementations **110**

6 Time and Space Lower Bounds for Nonblocking Implementations **111**
6.1 The Lower Bound . . . 111
6.1.1 The Intuition . . . 111

6.1.2 Perturbable Types 113
6.1.3 The Main Result 114
6.2 Examples of Perturbable Types 118
6.2.1 Modulo Counter and Related Objects 118
6.2.2 Compare&Swap 120
6.2.3 LL/SC Bit 122
6.2.4 Single Writer Snapshot 124
6.3 Applications 125

List of Figures

2.1 Specification of register r supporting {LL, SC, VL, read, write} by process p_i $(1 \leq i \leq n)$ 10

3.1 Examples of trees 20
3.2 A snapshot of data structure 21
3.3 Example of a process making progress 23
3.4 Another example of a process making progress 24
3.5 The lists for the trees in Figure 3.2 25
3.6 Unbounded construction for closed object $\mathcal{O}$ (Figures 3.6 to 3.9) 31
3.7 Unbounded construction for closed object $\mathcal{O}$ (Figures 3.6 to 3.9) 32
3.8 Unbounded construction for closed object $\mathcal{O}$ (Figures 3.6 to 3.9) 33
3.9 Unbounded construction for closed object $\mathcal{O}$ (Figures 3.6 to 3.9) 34

4.1 BSC* construction for closed object $\mathcal{O}$ (Figures 4.1 to 4.5) 64
4.2 BSC* construction for closed object $\mathcal{O}$ (Figures 4.1 to 4.5) 65
4.3 BSC* construction for closed object $\mathcal{O}$ (Figures 4.1 to 4.5) 66
4.4 BSC* construction for closed object $\mathcal{O}$ (Figures 4.1 to 4.5) 67
4.5 BSC* construction for closed object $\mathcal{O}$ (Figures 4.1 to 4.5) 69
4.6 Selecting a valid cell 87
4.7 Modifying BSC* to get BSC implementation 88

5.1 Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9) 95
5.2 Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9) 96
5.3 Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9) 97
5.4 Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9) 98
5.5 Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9) 99
5.6 Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9) 100
5.7 Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9) 101
5.8 Contention-sensitive construction for closed objects: `getToken` (Figures 5.1 to 5.9) 103
5.9 Contention-sensitive construction for closed objects: (Figures 5.1 to 5.9) 104

6.1 Statement S_k 114

Chapter 1

Introduction

1.1 Background

In an age of pervasive influence of the Internet on every aspect of the global society, the role of distributed computation is greater than ever. With vast numbers of computers connected over the Internet, algorithms that efficiently synchronize concurrent accesses to shared data resources by widely dispersed computers will be increasingly important. This thesis studies such algorithms in the more modest setting of shared memory systems.

A widely accepted method of synchronizing concurrent accesses of a data resource is by the use of *locks*. A process that wants to access the data resource first acquires the lock; when the process has completed using the data resource, it releases the lock. Since there is at most one process holding the lock at any time, this approach enforces sequential, exclusive access to the data resource. With this approach of *lock-based synchronization*, a process P has to wait for the process Q that currently holds the lock to release the lock. The progress of P therefore depends on the speed of Q. Furthermore, if Q crashes while holding the lock, P has no way of making progress.

In this thesis, we study *wait-free synchronization*. In contrast to lock-based synchronization, wait-free synchronization allows any process P to access a data resource without waiting for any other process to perform any action. Consequently, the progress of P is completely independent of the progress of any other process. Even if all other processes have crashed, P can still make progress.

We explain below the model of computation, the basic concepts, and the terminology.

1.1.1 Shared Memory System

We consider a *shared memory system*, where n processes communicate through shared memory. The system is *asynchronous*: There is no global clock that governs the speed of the processes in the system. Therefore, a process may be arbitrarily fast, or slow, in taking steps to either access shared memory, or perform local computation.

Shared registers in the memory support concurrent access by n processes. An operation on a shared register is *atomic*, i.e. the operation appears to take effect at one instant in time, even

though in reality it occupies a time interval, and may therefore overlap with other operations on the same register.

A shared register supports *read* and *write* operations. In addition, depending on the architecture, it may support some synchronization instructions, such as *test&set*, *fetch&add*, *compare&swap*, LL (*load-link*) and SC (*store-conditional*).[1]

1.1.2 Shared Objects and their Implementation

In a shared memory system, processes need to access shared data structures, files and databases, either in the course of their computation, or as a means of communicating and synchronizing with other processes. Clearly, such *shared objects* are essential in such a system.

Shared objects include registers (which are usually implemented in hardware) and shared data structures, such as queues, stacks, counters, heaps, files, databases (which must be implemented in software). The *type* of a shared object $\mathcal{O}$ specifies $\mathcal{O}$'s behavior when operations are applied sequentially (without overlap) to $\mathcal{O}$. Examples of types are: register supporting *read* and *write*; register supporting *compare&swap*, *read* and *write*; queue; and counter.

The following are some examples of software implementations of shared objects: implementing a shared queue, or a shared counter, using shared registers that support *read*, *write* and *compare&swap*; implementing a shared register that supports *read*, *write* and *compare&swap*, using shared registers that support *read*, *write*, LL and SC. The object being implemented is called the *implemented object*. The shared objects used in the implementation are called *base objects*.

1.1.3 Lock-based, Wait-free, and Nonblocking Implementation

The common approach to implementing shared data objects is based on locking: To perform an operation on a shared object, a process obtains a lock, accesses the object, and then releases the lock. Locking, however, has several drawbacks, including convoying (a descheduled process that holds a lock causes other processes to wait), priority inversion (a low priority process holds a lock needed by a high priority process, and the low priority process is preempted by a medium priority process), and deadlocks (each of two processes waits for a lock currently held by the other). Locking also limits parallelism: even when operations update disjoint parts of the data structure, they are applied sequentially, one after the other. Finally, lock-based implementations are not fault-tolerant: if a process crashes while holding a lock, other processes can end up waiting forever for the lock.

Wait-free implementations were conceived to overcome most of the above drawbacks of locking [Lam77, Her91]. A wait-free implementation guarantees that if a process repeatedly takes steps, then its operation on the implemented data structure will eventually complete, *regardless* of whether other processes are slow, fast, or have crashed. By definition, wait-free implementations are free of convoying, priority inversion, deadlocks, and are also resilient to process crashes. A weaker form of implementation, known as *nonblocking implementation* [Lam77], guarantees that if a process P repeatedly takes steps, then the operation of *some* process (not necessarily P) will

[1] These instructions are defined in Section 2.2.

eventually complete. Thus, nonblocking implementations guarantee that the system as a whole makes progress, but admit starvation of individual processes.

The locking and the lock-free (nonblocking/wait-free) approaches offer different trade-offs: while lock-based implementations are susceptible to delays and lack fault-tolerance, lock-free implementations tend to have higher latencies in practice. While lock-based implementations are in widespread use in all computer systems, lock-free implementations hold great promise for the future.

1.1.4 Linearizability

Whether implementations are lock-based or lock-free, they must ensure that concurrent operations are *linearizable*, *i.e.* they appear to take effect in some serial order (no operation may see the "partial effects" of another operation) [HW90]. Locking achieves this correctness condition by explicitly serializing accesses to the data structure. Herlihy proved a fundamental result that this correctness condition can also be achieved in wait-free implementations [Her91]. Specifically, he presented a *universal construction*—an algorithm that transforms the sequential implementation of *any* data structure into a wait-free implementation that multiple processes can concurrently access.

1.1.5 Deterministic and Randomized Implementation

In a *deterministic* implementation of a shared object, the next step that a process takes is determined entirely by the current state of the system. On the other hand, in a *randomized* implementation of a shared object, the next step of a process may additionally depend on the outcome of a random event, such as a coin toss.

1.2 Contributions of the Thesis

This thesis consists of two parts. In Part One, we present an efficient wait-free implementation of a class of object types; In Part Two, we prove time and space lower bounds on wait-free implementations of another class of object types. (A preliminary version of Part One appears in [CJT98]. Part Two was first published in a preliminary version in [JTT96], and subsequently in a journal [JTT00].) The following is a brief summary of our results. (In the following subsections, we provide detailed descriptions of our contributions and their significance.)

- We identify a class of object types that we call *closed types*: Closed types include several useful synchronization types, such as *swap* and *fetch&add*. We present a wait-free algorithm that efficiently implements any closed type, with a polylogarithmic worst-case time complexity. We believe that ours is the first wait-free algorithm that implements a class of types (as opposed to algorithms that implement a specific type) to achieve a sub-linear worst-case time complexity.

- We identify a class of object types that we call *perturbable types*: Perturbable types include several well-known types, such as *fetch&add*, *LL/SC bit* and *compare&swap*. We prove a linear lower bound on both the space and time complexity of any wait-free implementation of

a perturbable type, where such an implementation uses only historyless objects and consensus objects. Our result improves on some previously known $\Omega(\sqrt{n})$ space complexity lower bounds. It also shows the near space-optimality of some known wait-free implementations.

1.2.1 A Polylog Time Wait-Free Construction for Closed Objects

In this section, we state our first result in detail. The design of wait-free implementations is intellectually complex because of the need to simultaneously satisfy linearizability and wait-freedom. Thus, it takes a great deal of effort to design an efficient wait-free implementation of every type of useful shared object. To address this difficulty, Herlihy proposed the notion of a *universal construction*, which we briefly describe below [Her91]. Universal constructions have since received a lot of research attention [ADT95, AMTT97, AM95a, AM95b, AD96, Bar93, Her88, Her91, Her93, IR94, JT92, Moi97b, Plo89, ST95, TSP92].

An *n-process universal construction* is an algorithm that takes as parameter the transition function of any type,[2] and has the following interesting property: if the parameter is bound to the transition function of any type T, the algorithm becomes a wait-free implementation of a type T object that can be accessed concurrently by n processes [Her91]. Thus, once we have an efficient universal construction U, any type of shared object can be efficiently implemented simply by passing the right parameter to U.

Unfortunately, the worst-case time complexity of every existing n-process universal construction is $\Omega(n)$. In fact, for a fairly large class of universal constructions, namely, *oblivious* universal constructions,[3] $\Omega(n)$ is a lower bound on the worst-case time complexity: to complete a single operation on any shared object implemented using any oblivious universal construction, in the worst case a process must perform $\Omega(n)$ local computation [Jay98a].

Since universal constructions with sub-linear time complexity do not seem possible, it is natural to seek a sub-linear time "semi-universal" construction that can implement a large class of types (as opposed to *all* types). We present such a construction in this thesis. Our contribution is described in the next subsection.

1.2.1.1 Our Contribution

We present a construction that implements a large class of types, which we call *closed types*. Informally, a type is closed if, for every pair (op', op'') of operations of the type, there is another operation op of the type such that executing op has the same effect on the state as executing op' followed by op''. For example, the type supporting *write* and *fetch&add* operations is closed because (1) *fetch&add*(a) and *fetch&add*(b) combine to *fetch&add*$(a+b)$, (2) *write*(a) and *fetch&add*(b) combine to *write*$(a+b)$, and (3) for any operation op, op and *write*(b) combine to *write*(b). We will give many more examples of closed types in Chapter 3. The highlight of our construction is that it has a polylogarithmic worst-case time complexity and is also adaptive, as described below.

[2] For each state s and operation op, the transition function defines the response and the new state that result from applying op in state s.

[3] Roughly speaking, a universal construction is *oblivious* if it does not exploit the structure of the transition function that its parameter is bound to (by, for instance, providing different implementations for different classes of transition functions).

Let $\mathcal{O}$ be a shared object implemented using our construction. The *contention* (on $\mathcal{O}$) at time t is the number of operations executing on $\mathcal{O}$ at time t. Let OP denote an execution of an operation on $\mathcal{O}$. The *contention experienced by* OP is the maximum contention during the interval in which OP executed. If n is the maximum number of processes that the construction is designed for and n_c is the contention experienced by OP, our construction guarantees that OP terminates in $O(\min(n_c, \log^2 n))$ steps. Thus, when contention is low, the time complexity depends only on the actual number n_c of processes contending simultaneously, rather than the maximum number n of processes that the construction is designed to handle. (Such constructions where the time complexity depends on contention, and not on n, are known as *adaptive* constructions [ADT95].) Furthermore, at *all* levels of contention, the time complexity is bounded by a small value, namely, $O(\log^2 n)$.

Jayanti characterized a class of types and proved a lower bound of $\Omega(\log n)$ on the worst-case time complexity of any implementation of a type from that class [Jay98b]. That class, it turns out, contains some closed types. It follows that the worst-case time complexity of $O(\log^2 n)$ achieved by our construction is within a logarithmic factor of the optimal.

To the best of our knowledge, if we consider constructions that can implement a class of types (as opposed to constructions that implement one specific type), this is the first time that the linear time barrier has been broken for the worst-case time complexity.

1.2.1.2 Hardware support

Our construction assumes that LL and SC operations can be performed on shared memory words.[4] Real machines do not directly support LL and SC operations, but this is not a problem because there are constant time implementations [Moi97a, JP03] of LL/SC operations from hardware operations supported by modern architectures, specifically the compare&swap operation supported by UltraSPARC [WG] and Itanium [Int02] and the "realistic LL/SC" operations (weak LL/SC operations, with spurious failures) supported by POWER4 [TDF$^+$01], MIPS [Sys02] and Alpha [Sit92]).

1.2.2 Time and Space Lower Bounds for Nonblocking Implementations

In this section, we state our second result in detail. Nonblocking and wait-free implementations of shared objects have been the subject of much research. While there have been several results on when such implementations are feasible and when they are not, results establishing their intrinsic time and space requirements are relatively scarce, especially for randomized implementations. In this thesis, we present a technique by which one can obtain a linear lower bound on the space complexity of several randomized nonblocking implementations. The technique also yields a linear lower bound on the time complexity of several deterministic nonblocking implementations.

Specifically, our results are as follows. Let $\mathcal{I}$ be any randomized nonblocking n-process implementation of any object in set A from any combination of objects in set B, where $A = \{$*increment*, *fetch&add*, modulo k *counter* (for any $k \geq 2n$), LL/SC bit, k-valued *compare&swap* (for any $k \geq n$), single-writer *snapshot*$\}$, and $B = \{$resettable consensus$\} \cup \{$historyless objects$\}$. (Roughly

[4]Since closed types include some universal types (see Example 5 of Section 2.3), a construction such as ours that implements closed types will necessarily require support for universal instructions, such as LL and SC.

speaking, an object is *historyless* if each of its operations either does not affect the state of the object or overwrites the previously applied operations. Examples include registers, *test&set* objects, and *swap* registers.) The space complexity of $\mathcal{I}$ is at least $n-1$. Moreover, if $\mathcal{I}$ is deterministic, both its time and space complexity are at least $n-1$. These lower bounds hold even if objects used in the implementation are of unbounded size.

Some of our lower bounds improve known lower bounds, while others are completely new. In particular, Fich, Herlihy & Shavit proved a $\Omega(\sqrt{n})$ space complexity lower bound for a randomized nonblocking n-process implementation of *binary consensus* from historyless objects [FHS93, FHS98]. Using this result, they showed that any randomized nonblocking n-process implementation of *compare&swap*, or *fetch&add*, or *bounded-counter* from historyless objects requires $\Omega(\sqrt{n})$ instances of such objects. Our results on *compare&swap*, *fetch&add*, and *bounded-counter* are stronger in two ways: (i) we show that at least $n-1$ objects are necessary, and (ii) we show that $n-1$ objects are needed even if the implementation is free to use *resettable consensus objects*, besides the historyless objects allowed by [FHS93, FHS98]. On the other hand, our lower bound technique applies only to implementations of "multiple-use" objects in which each process can access the implemented object many times. In contrast, the technique of Fich, Herlihy, and Shavit applies even to implementations of "single-use" objects. Thus, our proof technique does not (and cannot) improve on the main result of Fich, Herlihy & Shavit, namely, their $\Omega(\sqrt{n})$ space complexity lower bound for a randomized nonblocking n-process implementation of binary consensus.

Our result also implies that the following deterministic implementations in the literature are almost space-optimal.

1. Afek et al. give two wait-free implementations of a *single writer snapshot object* consisting of n segments, each one written by a different process: one from *unbounded registers* and one from *bounded registers* [AWW93]. The one that uses unbounded registers is of space complexity n. We prove a lower bound of $n-1$.

2. Aspnes gives a wait-free implementation of an n-process *bounded-counter* from a single instance of a *single writer snapshot object* [Asp90]. Combined with the above result of Afek et al., this implies that bounded-counter can be implemented from n unbounded registers. We prove that at least $n-1$ *registers* are necessary when the bounded-counter is a modulo k counter, where $k \geq 2n$.

In both cases above, the lower bound of $n-1$ is particularly appealing because it applies to even randomized nonblocking implementations while the upper bound of n holds for deterministic wait-free implementations.

In fact, the lower bounds proved here (and the lower bounds in [FHS93, FHS98]) apply not just to nonblocking implementations, but to any implementation satisfying a weaker progress condition called *solo-termination*, defined in [FHS98]. Roughly speaking, a deterministic implementation is solo-terminating if at every configuration C in a system execution the following holds for all processes p: if p runs alone from C, p's operation on the implemented object will eventually complete. For a randomized implementation to be considered solo-terminating we require that for all C and p, if p runs alone from C there is at least one sequence of outcomes for p's coin tosses that will enable p to complete its operation on the implemented object.

It is well-known that a wait-free implementation is also nonblocking. It is clear that a nonblocking implementation is also solo-terminating. Thus, the lower bounds that we prove here for solo-terminating implementations apply also to nonblocking and wait-free implementations.

There is a large body of research on algorithms for synchronous parallel computers (such as the PRAM model, the mesh, perfect shuffle and hypercube architectures) that has resulted in many algorithms whose time complexity is polylogarithmic in the number of processors. In contrast, for the asynchronous model of computing, wait-free algorithms of polylog complexity are rare ([Cha96, Aum97] and the algorithm in Part One of this thesis ([CJT98]) are some notable exceptions). In fact, our lower bound implies that for deterministic wait-free implementations of many common objects from certain base objects, there are no algorithms of sub-linear time complexity, let alone polylog complexity.

Our proof technique for the lower bounds is general in that we have successfully applied it to implementations of a variety of objects. The technique is also interesting because (i) it simultaneously yields a lower bound on time and space complexities of implementations, (ii) the lower bound on space complexity holds even for randomized implementations, and (iii) the lower bounds apply not just to nonblocking or wait-free implementations, but to any implementation satisfying the weaker solo-termination progress condition.

1.3 Organization of the Thesis

The model of computation is presented in Chapter 2.

Part One of this thesis consists of three chapters: Chapter 3 presents an unbounded implementation of closed objects that requires unbounded shared memory. Chapter 4 presents a bounded implementation, based on the unbounded implementation. Chapter 5 describes the enhancement to the bounded implementation that is needed to make its time complexity contention-sensitive.

Part Two of the thesis is in Chapter 6, where we prove the lower bounds on nonblocking and wait-free implementations of perturbable objects.

Chapter 2

Model

In this chapter, we define the model of computation and the shared object types that we encounter in this thesis.

2.1 Object Type

An *object type* T is a tuple $(OP, RES, Q, \delta_{state}, \delta_{resp})$, where OP is a set of operations, RES is a set of responses, Q is a set of states, $\delta_{state} : Q \times OP \rightarrow Q$ is a state transition function, and $\delta_{resp} : Q \times OP \rightarrow RES$ is a response function. Informally, if $\delta_{state}(\sigma, op) = \sigma'$ and $\delta_{resp}(\sigma, op) = res$, applying operation *op* to an object in state σ causes the object to move to state σ' and return the response res. $(\delta_{state}, \delta_{resp})$ is known as the *sequential specification* of type T.

2.2 Common Object Types

We provide the definitions of some common object types below.

2.2.1 Test&Set

A *test&set* object $\mathcal{O}$ is a bit that supports two operations: *test&set* and *reset*. The operation *test&set* sets the state to 1, and returns the old state of $\mathcal{O}$. The operation *reset* sets the state to 0, and returns *ack*.

2.2.2 Swap

A *swap* object $\mathcal{O}$ is a register that supports *read* and *swap*. The operation *read* has the standard semantics. We now define *swap* (u). Let the old state of $\mathcal{O}$ be w, then *swap* (u) causes the new state to become u, and returns w.

2.2.3 Compare&Swap

A *compare&swap* object $\mathcal{O}$ is a register that supports *read*, *write*, and *compare&swap*. The operations *read* and *write* have the standard semantics. We now define *compare&swap* (u, v). If

the old state of $\mathcal{O}$ is u, then the new state after *compare&swap* (u, v) is v, and the operation returns *true*. If the old state of $\mathcal{O}$ is not u, then the state after *compare&swap* (u, v) is unchanged, and the operation returns *false*.

2.2.4 Fetch&Add

A *fetch&add* object $\mathcal{O}$ is a register that supports *read*, *write*, and *fetch&add*. The operations *read* and *write* have the standard semantics. We now define *fetch&add* (a). If the old state of $\mathcal{O}$ is x, then the new state after *fetch&add* (a) is $x + a$, and the operation returns x.

2.2.5 Increment

An *increment* object is a restricted form of a *fetch&add* object that supports only *read*, *write*, and *fetch&add* (1).

2.2.6 Fetch&Multiply

A *fetch&multiply* object $\mathcal{O}$ is a register that supports *read*, *write*, and *fetch&multiply*. The operations *read* and *write* have the standard semantics. We now define *fetch&multiply* (a). If the old state of $\mathcal{O}$ is x, then the new state after *fetch&multiply* (a) is $x \times a$, and the operation returns x.

2.2.7 Fetch&Φ

A *fetch&*Φ object $\mathcal{O}$ is a register that supports *read*, *write*, and *fetch&*Φ, where Φ is the logical *and*, *or*, or *complement*. The operations *read* and *write* have the standard semantics. Let the old state of $\mathcal{O}$ be x. *fetch&and* (a) causes the new state to become $x \wedge a$ (bitwise conjunction), and returns x. *fetch&or* (a) causes the new state to become $x \vee a$ (bitwise disjunction), and returns x. *fetch&complement* causes the new state to become the bitwise complement of x, and returns x.

2.2.8 LL, SC, and VL

Our construction for closed objects (in Part One) uses shared registers that support LL, SC, VL, *read* and *write* as base objects. The formal specification of such a register r is given in Figure 2.1. We now describe the behavior of r. The state of r is a pair (value(r), Pset(r)), where value(r) is the value of the register r, and Pset(r) is a set of processes to be defined shortly. An *update* to value(r) takes place whenever an operation writes successfully to r. Thus, it is possible that an update to value(r) does not change the value of r. Pset(r) is the set of processes that have executed a LL operation on r since the most recent update to value(r).

Suppose p_i executes LL(r) when value(r)=x, Pset(r) = S. Then, after LL(r), value(r)=x, Pset(r)=$S \cup \{P\}$, and x is the value returned.

Suppose p_i executes VL(r) when value(r)=x, Pset(r) = S. VL(r) does not change the state of r. VL(r) returns *true* if $P \in S$, and returns *false* otherwise. Thus, VL(r) indicates whether value(r) has been updated since p_i's most recent LL(r).

Suppose p_i executes SC(r, v) when value(r)=x, Pset(r) = S. The operation SC(r, v) *succeeds* if $P \in S$; otherwise, SC(r, v) *fails*. (SC(r, v) succeeds if value(r) has not been updated since p_i's

LL(r)	SC(r, v)	read(r)
Pset(r) := Pset(r) $\cup \{p_i\}$	**if** $p_i \in$ Pset(r)	**return** value(r)
return value(r)	value(r) := v	
	Pset(r) := $\emptyset$	write(r, v)
VL(r)	**return** *true*	value(r) := v
if $p_i \in$ Pset(r) **return** *true*	**else return** *false*	Pset(r) := $\emptyset$
else return *false*		

Figure 2.1: Specification of register r supporting {LL, SC, VL, read, write} by process p_i ($1 \leq i \leq n$)

most recent LL(r); SC(r, v) fails otherwise.) If SC(r, v) succeeds, then the new state of r is $(v, \emptyset)$, *i.e.* the new value of r is v, and Pset(r) is set to $\emptyset$ (because value(r) has just been updated). A successful SC(r, v) returns *true*. If SC(r, v) fails, then the state of r is unchanged, and SC(r, v) returns *false*.

A *write* operation *write*(r, v), or equivalently, $r := v$, changes the state of r to $(v, \emptyset)$, *i.e.* the new value of r is v, and Pset(r) is set to $\emptyset$ (because value(r) has just been updated). A *read* operation on r does not change the state of r, and returns value(r).

2.2.9 Resettable Consensus

We now define the type `resettable consensus`. The state of $\mathcal{O}$ is either $\perp$ or a natural number. $\mathcal{O}$ supports three operations: *propose* v, where v is a natural number, *read*, and *reset*. *read* simply returns the current state of $\mathcal{O}$, leaving the state unchanged. *reset* sets the state of $\mathcal{O}$ to $\perp$, and returns *ack*. The behavior of $\mathcal{O}$ when *propose* v is applied depends on the old state of $\mathcal{O}$: If the old state is $\perp$, the new state becomes v, and v is returned. If the old state is u (some natural number), then the state is unchanged, and u is returned. (In either case, the value returned is the first value to be proposed after the most recent *reset*.)

Formally, the type `resettable consensus` is $(OP, RES, Q, \delta_{state}, \delta_{resp})$, where

- $OP = \{\text{read}, \text{reset}\} \cup \{\text{propose } v \mid v \in \mathcal{N}\}$, where $\mathcal{N}$ is the set of natural numbers
- $RES = \mathcal{N} \cup \{ack\}$,
- $Q = \mathcal{N} \cup \{\perp\}$
- $\delta_{state}, \delta_{resp}$ are as follows:
  - For all $u \in Q$, $\delta_{state}(u, \text{read}) = u$, $\delta_{resp}(u, \text{read}) = u$;
  - For all $u \in Q$, $\delta_{state}(u, \text{reset}) = \perp$, $\delta_{resp}(u, \text{reset}) = ack$;
  - $\delta_{state}(\perp, \text{propose } v) = v$, $\delta_{resp}(\perp, \text{propose } v) = v$;
  - For all $u \in \mathcal{N}$, $\delta_{state}(u, \text{propose } v) = u$, $\delta_{resp}(u, \text{propose } v) = u$.

Resettable consensus was first defined by Herlihy [Her88, Her91], but included only the propose and reset operations. We added the read operation to make our lower bound result stronger. Our definition of resettable consensus is similar to the sticky bit defined by Plotkin [Plo89].

2.3 Closed and Closable Types

We say that operations op' and op'' *combine to* op, denoted by $op' \bigotimes op'' = op$, if for all $s \in Q$ we have $\delta_{state}(s, op) = \delta_{state}(\delta_{state}(s, op'), op'')$. That is, from any state s, applying op results in the same state as first applying op' and then applying op''. (There is no constraint on $\delta_{resp}(s, op)$.) We note that $\bigotimes$ is associative: $(op_1 \bigotimes op_2) \bigotimes op_3 = op_1 \bigotimes (op_2 \bigotimes op_3)$. The type T is *closed* if, for all ordered pairs $(op', op'') \in OP \times OP$, there exists $op \in OP$ such that $op' \bigotimes op'' = op$.

We restrict our focus to closed types for which the functions δ_{state}, δ_{resp}, and $\bigotimes$ can be computed in $O(1)$ time, and a state or an operation can be stored in a small constant number of machine words.

A type $T' = (OP', RES', Q', \delta'_{state}, \delta'_{resp})$ is a *super type* of $T = (OP, RES, Q, \delta_{state}, \delta_{resp})$ if $OP' \supseteq OP, RES' \supseteq RES, Q' \supseteq Q$ and, for all $(s, op) \in Q \times OP$, we have $\delta'_{state}(s, op) = \delta_{state}(s, op)$ and $\delta'_{resp}(s, op) = \delta_{resp}(s, op)$. A type $T = (OP, RES, Q, \delta_{state}, \delta_{resp})$ is *closable* if it has a closed super type $T' = (OP', RES', Q', \delta'_{state}, \delta'_{resp})$ of T. Each operation in $OP' - OP$ is called a *closing operation* of T.

Notice that whenever a shared object of a closable type T is desired, one can instead implement a shared object of a closed super type of T. Thus, the construction presented in this paper for closed types, is also good for implementing closable types. The following are examples of closed or closable types (some examples are taken from [KRS86]):

1. The *test&set* object is clearly closed.

2. The type supporting *swap* and *read* operations is trivially closed.

3. The type supporting *fetch&add* (*F&A*), *read*, and *write* operations is closed, as verified below:

   - $F\&A(a) \bigotimes F\&A(b) = F\&A(a + b)$
   - $write(a) \bigotimes F\&A(b) = write(a + b)$
   - For any operation op, $op \bigotimes write(a) = write(a)$
   - For any operation op, $op \bigotimes read = read \bigotimes op = op$

4. Consider the type supporting *read* and *all* of the operations in the *fetch&*Φ family, where Φ is the logical *and*, *or*, or *complement*.

   We claim that the above type is closable: to close the type, add the operation *closing-op*(a, b, f), where the arguments a and b are bit vectors (of the same length as the state) and f is a boolean. This operation changes the current state x as follows: if f is *false*, the new state is $(a \wedge x) \vee b$; otherwise the new state is $(a \wedge \overline{x}) \vee b$, where $\wedge$ and $\vee$ are bitwise logical operations, and $\overline{x}$ is the bitwise complement of x.

To see that this type is closed, we first observe that, with respect to the resulting state, *fetch&and*, *fetch&or*, and *fetch&complement* are all instances of *closing-op*(a, b, f). For example, *fetch&and*(c) = *closing-op*$(c, 0, false)$, where 0 is the vector 00...00. It suffices therefore to show that two *closing-op*s combine to a *closing-op*. Since a *closing-op* consists of three component logical operations: *complement* (optional, depending on the third parameter of the *closing-op*), *and*, and *or*, we need to show only that *closing-op*$(a, b, f) \bigotimes$ *fetch&and*(c), *closing-op*$(a, b, f) \bigotimes$ *fetch&or*(c), *closing-op*$(a, b, f) \bigotimes$ *fetch&complement* are *closing-op*s. This is indeed the case, as shown by the following easily verified facts:

- *closing-op*$(a, b, f) \bigotimes$ *fetch&and*(c) = *closing-op*$(a \wedge c, b \wedge c, f)$
- *closing-op*$(a, b, f) \bigotimes$ *fetch&or*(c) = *closing-op*$(a, b \vee c, f)$
- *closing-op*$(a, b, f) \bigotimes$ *fetch&complement* = *closing-op*$(\overline{b}, \overline{a \vee b}, \overline{f})$

With these facts, we see that an object that supports *read* and *all* of the operations in the *fetch&*Φ family is indeed closable.

5. The type supporting all of *read*, *swap*, *fetch&add* and *fetch&multiply*. Here again we close the type by adding the operation *closing-op*(a, b) which, when applied in state x, changes the state to $ax + b$. We note that, with respect to the resulting state, *fetch&add* and *fetch&multiply* are instances of *closing-op*(a, b). With this *closing-op*, the type is closed, as verified below:

   - *closing-op*$(a, b) \bigotimes$ *closing-op*(c, d)= *closing-op*$(ac, bc + d)$
   - *swap*$(a) \bigotimes$ *closing-op*(c, d)= *swap*$(ac + d)$
   - For any operation *op*, *op* $\bigotimes$ *swap*(a) = *swap*(a)
   - For any operation *op*, *op* $\bigotimes$ *read* = *read* $\bigotimes$ *op* = *op*

6. The type whose state consists of a pair of values (x, y), and supports the following operations (for each operation, the state immediately before the operation is assumed to be (x, y)): *move*$_1$ changes state to (x, x), *move*$_2$ changes state to (y, y), *swap* changes state to (y, x), *write*$_1(z)$ changes state to (z, y), *write*$_2(z)$ changes state to (x, z), *write*(z, z') changes state to (z, z'), and *read* returns (x, y).

   We define the following *closing-op*(l, r), where the value of l, and r is either X, Y, (X, Y are special symbols), or some valid value v. Let the state of the object before *closing-op*(l, r) be (x, y). Let v be a valid value. Then, the state of the object after *closing-op*(l, r) is defined as follows: (* denotes that the component is not defined.)

   - the state of the object after *closing-op*$(X, *)$ is $(x, *)$.
   - the state of the object after *closing-op*$(*, X)$ is $(*, x)$.
   - the state of the object after *closing-op*$(Y, *)$ is $(y, *)$.
   - the state of the object after *closing-op*$(*, Y)$ is $(*, y)$.
   - the state of the object after *closing-op*$(v, *)$ is $(v, *)$.

- the state of the object after *closing-op*$(*, v)$ is $(*, v)$.

It is easy to verify that all the operations, such as $move_1, write_1$, are instances of *closing-op*(l, r). The rules for combining two *closing-op*s are:

- *closing-op*$(e, f) \bigotimes$ *closing-op*$(X, *)$ =*closing-op*$(e, *)$.
- *closing-op*$(e, f) \bigotimes$ *closing-op*$(*, X)$ =*closing-op*$(*, e)$.
- *closing-op*$(e, f) \bigotimes$ *closing-op*$(Y, *)$ =*closing-op*$(f, *)$.
- *closing-op*$(e, f) \bigotimes$ *closing-op*$(*, Y)$ =*closing-op*$(*, f)$.
- *closing-op*$(e, f) \bigotimes$ *closing-op*$(v, *)$ =*closing-op*$(v, *)$.
- *closing-op*$(e, f) \bigotimes$ *closing-op*$(*, v)$ =*closing-op*$(*, v)$.

Therefore, our type is closable. Unlike previous examples, this type is universal [Her91]. [1]

Thus, there is a lot of variety among closed and closable types: there are commutative, overwriting, non-commutative and non-overwriting, and universal types. Further, as the examples have shown, this class includes many commonly used synchronization objects. Our construction implements all of these in polylog time.

2.4 Historyless Types

Let $op(\sigma)$ denote $\delta_{state}(\sigma, op)$. The following definitions are from Fich et al [FHS98]. An operation op is *trivial* if its application does not affect the state; that is, for all states σ, $op(\sigma) = \sigma$. Operation op' *overwrites* operation op if applying op and then op' results in the same state as simply applying op'; more precisely, for all states σ, $op'(op(\sigma)) = op'(\sigma)$. A type is *historyless* if all its nontrivial operations overwrite one another. A *test&set* object, and a register that supports *read*, *write* and *swap* are examples of historyless types.

Proposition 1 *For a historyless type, the following statements are true:*

1. *For all states σ, nontrivial operations op_k and finite sequences $op_{k-1} \cdots op_1$ of operations,* $op_k(op_{k-1}(\cdots op_1(\sigma) \cdots)) = op_k(\sigma)$.
2. *For all states σ and finite sequences $op_k op_{k-1} \cdots op_1$ of trivial operations,* $op_k(op_{k-1}(\cdots op_1(\sigma) \cdots)) = \sigma$.

Proof By a simple induction on the length of the operation sequence. □

[1] Universal types are defined in Section 2.7.

2.5 Implementation

A *randomized implementation* is specified by the following elements:

- the type and the initial state of the implemented object $\mathcal{O}$ (the initial state of $\mathcal{O}$ is a state of its type).
- a set of objects $O_1, \ldots, O_m$ from which $\mathcal{O}$ is implemented, their types and their initial states.
- a set of processes $p_1, p_2, \ldots, p_n$ that may access $\mathcal{O}$. (For notational convenience, the processes are named $0, 1, \ldots, n-1$, instead of $p_1, p_2, \ldots, p_n$, in Part One of this thesis.)
- a set of *randomized access procedures* $\texttt{apply}(p_i, op, \mathcal{O})$, for $1 \leq i \leq n$ and $op \in OP$, where OP is the set of operations associated with the type of $\mathcal{O}$.

The access procedure $\texttt{apply}(p_i, op, \mathcal{O})$ specifies how p_i should execute the operation *op* on $\mathcal{O}$ in terms of operations on $O_1, \ldots, O_m$. The value returned by the procedure is deemed to be the response from $\mathcal{O}$. We call $O_1, \ldots, O_m$ the *base objects* of the implementation. The *space complexity* of the implementation is m.

We consider a system that consists of processes $p_1, \ldots, p_n$ and an implemented object $\mathcal{O}$ that $p_1, \ldots, p_n$ may access. We denote such a system by $(p_1, \ldots, p_n; \mathcal{O})$. Each p_i has a set of states and has a distinguished input variable $op\text{-}list_i$. This variable is an infinite sequence of operations $op_1, op_2, \ldots$ where each op_j is an operation supported by $\mathcal{O}$. [2] Each p_i performs the following actions repeatedly forever: obtain the next operation *op* from $op\text{-}list_i$ and execute the access procedure $\texttt{apply}(p_i, op, \mathcal{O})$ until it returns.

Let COINSPACE be a non-empty countable set of all possible outcomes of a coin toss, where the probability of each outcome in the set is non-zero, and the sum of the probabilities of all outcomes is one. The *state* of p_i consists of a pointer to the operation *op* in $op\text{-}list_i$ that p_i is currently executing on $\mathcal{O}$, its program counter (*i.e.* the Line of $\texttt{apply}(p_i, op, \mathcal{O})$ that it is executing), and the values of p_i's local registers (as specified by $\texttt{apply}(p_i, op, \mathcal{O})$). Process p_i executes $\texttt{apply}(p_i, op, \mathcal{O})$ in *steps*. Each step consists of the following sequence of actions, all of which occur together atomically:

- p_i tosses a coin. (All coin tosses are independent.) Let *toss-outcome* $\in$ COINSPACE denote the outcome of this toss.
- *toss-outcome* and p_i's current state uniquely determine an operation *oper* and a base object O_j that *oper* should be applied to. Accordingly, p_i performs some local computations, then applies *oper* to O_j.
- O_j changes state and returns a response. The new state of O_j and the response are uniquely determined by the sequential specification of O_j.

[2] $op\text{-}list_i$ or a finite prefix of $op\text{-}list_i$ is the sequence of operations that p_i applies on $\mathcal{O}$. The sequence of operations that p_i applies on $\mathcal{O}$ may be decided by p_i dynamically (*i.e.* it does not need to be fixed during initialization).

- The response from O_j, together with *toss-outcome* and p_i's current state, uniquely determine the local computations that p_i now performs, and the new state of p_i after these local computations. It is possible for the procedure $\texttt{apply}(p_i, op, \mathcal{O})$ to terminate, returning some response. In this case, the new state of p_i reflects both the fact that the access procedure terminated and the response returned by the access procedure. Further, p_i's step enabled from this state corresponds to the first step of the access procedure $\texttt{apply}(p_i, op', \mathcal{O})$, where op' is the operation that immediately follows op in $op\text{-}list_i$.

Thus, in one step, a process applies one operation on a base object, and performs some number of local computations.

A *deterministic implementation* is a special case of a randomized implementation for which COINSPACE, the set of possible outcomes for a coin toss, is a singleton set.

A *configuration* of $(p_1, \ldots, p_n; \mathcal{O})$ is a tuple $(\sigma_1, \ldots, \sigma_n, rem_1, \ldots, rem_n, \tau_1, \ldots, \tau_m)$, where σ_i is a state of p_i, rem_i is a suffix of $op\text{-}list_i$ (and corresponds to the infinite sequence of operations that p_i is yet to initiate on $\mathcal{O}$), and τ_j is a state of base object O_j. We note that the implementation specifies a unique initial state for each base object and a unique initial state for each process p_i. It follows that the initial configuration is uniquely determined by an assignment of infinite sequences of operations to the input variables $op\text{-}list_i$ $(1 \leq i \leq n)$. An *execution fragment from configuration* C_0 of $(p_1, \ldots, p_n; \mathcal{O})$ is a finite or infinite sequence $C_0, C_1, C_2, \ldots$ of configurations such that, for all $k \geq 0$, C_{k+1} is the configuration that results when some process performs a step in configuration C_k. An *execution*, or a *run*, is an execution fragment from an initial configuration.

A *schedule* is a finite or an infinite sequence $[p_{i_1}, t_1], [p_{i_2}, t_2], \ldots$ where each p_{i_j} is from $\{p_1, \ldots, p_n\}$ and each t_j is from COINSPACE. If C is a configuration and $\alpha = [p_{i_1}, t_1], [p_{i_2}, t_2], \ldots$ is a schedule, EXEC(C, α) denotes the execution fragment $C_0, C_1, C_2, \ldots$ where $C = C_0$ and each C_k results from C_{k-1} when p_{i_k} takes a step in which the outcome of p_{i_k}'s toss is t_k. A configuration C is *reachable* if there is some initial configuration C_0 and a finite schedule α such that the configuration at the end of EXEC(C_0, α) is C.

An implementation is *correct* if it has two properties—linearizability (safety property) and a liveness property (wait-freedom in Part One, solo-termination in Part Two). These properties are described next.

2.6 Linearizability

Let $\mathcal{O}$ be an implemented object shared by processes $p_1, \ldots, p_n$. Consider an execution E of $(p_1, \ldots, p_n; \mathcal{O})$ in which process p_i applies operation op on $\mathcal{O}$, *i.e.,* p_i invokes the procedure $\texttt{apply}(p_i, op, \mathcal{O})$. We say that op is *complete* in E if the procedure $\texttt{apply}(p_i, op, \mathcal{O})$ terminates in E. We say that op is *incomplete* in E if $\texttt{apply}(p_i, op, \mathcal{O})$ does not terminate in E. An execution E of $(p_1, \ldots, p_n; \mathcal{O})$ is *linearizable* if there exists a set S of all the complete operations and some (possibly all) of the incomplete operations in E, such that:

- for every operation op (say) by process p_i (say) in S, the operation appears to take effect at some instant during the execution of $\texttt{apply}(p_i, op, \mathcal{O})$ (in other words, operations in S appear atomic).

- no other operation takes effect.

The *implementation of $\mathcal{O}$ is linearizable* if every execution of $(p_1, \ldots, p_n; \mathcal{O})$ is linearizable. The order in which the operations in S take effect is called the *linearization order*.

2.7 Wait-freedom

An implementation whose access procedures never terminate is trivially linearizable. Such an implementation, however, is not likely to be useful. Thus, in addition to linearizability, implementations should guarantee certain progress properties. *Wait-freedom* and *nonblockingness* are the progress conditions that received the most attention recently [HS93]. In this thesis, we present only deterministic (and no randomized) wait-free algorithms. Thus, we define *wait-freedom* here in the restricted context of deterministic algorithms.

An implementation of object $\mathcal{O}$ is *wait-free* if the following holds: Let E be any execution in which process p_i takes infinitely many steps. Then every operation by p_i in E is complete.

A type T is *universal* [Her91] if it is possible to implement, wait-freely and deterministically, an object of *any* type using only registers that support *read* and *write*, and objects of type T.

2.8 Solo-termination

In Part Two, we consider a progress property for a randomized implementation that is weaker than wait-freedom and nonblockingness: it is called *solo-termination*, first defined in [FHS98]. An implementation has the solo-termination property if for each reachable configuration C and each process p the following holds: if p runs alone from configuration C, then there is at least one sequence of outcomes for p's coin tosses that will enable p to complete an operation on the implemented object. More precisely, a *randomized implementation of $\mathcal{O}$ is solo-terminating* if, for all reachable configurations C and all processes p_i, there is some finite schedule $\alpha = [p_i, t_1], [p_i, t_2], \ldots, [p_i, t_k]$ such that p_i completes an operation on $\mathcal{O}$ during $\text{EXEC}(C, \alpha)$. (Since the probability of each t_j is non-zero by definition, there is a non-zero probability of p_i completing an operation on $\mathcal{O}$ when p_i runs alone from C.)

The lower bounds proved in Part Two apply to solo-terminating (and therefore to nonblocking and wait-free) implementations.

2.9 Notation

For a schedule α, $|\alpha|$ denotes its length. We say α *contains process* p if, for some t, $[p, t]$ is in the sequence α. $\text{PSET}(\alpha)$ denotes the set of all processes contained in α. If α and β are any schedules, $\alpha\beta$ denotes the schedule which is the concatenation of α and β.

If Σ is a set, Σ^* denotes the set of all *finite* sequences of elements from Σ (including the empty sequence, denoted by ϵ). Notice that $(\{p_1, \ldots, p_n\} \times \text{COINSPACE})^*$ is the set of all finite schedules.

In our proofs in Part Two, when we consider a system $(p_1, \ldots, p_n; \mathcal{O})$ (where $\mathcal{O}$ is an object implemented for processes $p_1, \ldots, p_n$), we fix the initial configuration of the system at some value, say C_0, right at the beginning of the proof by specifying the initial values of the input variables

op-list$_i$ ($1 \leq i \leq n$). Since the initial configuration is fixed, each schedule $\alpha \in (\{p_1, \ldots, p_n\} \times \textsc{coinspace})^*$ uniquely determines the execution $\textsc{exec}(C_0, \alpha)$. Therefore, for brevity, if α is a schedule, we will use the same symbol α to also denote the execution $\textsc{exec}(C_0, \alpha)$. From the context it will be clear whether α refers to the schedule or to the execution. If α and β are schedules and S is a set of processes or a set of base objects, We write $\alpha \approx_S \beta$ if, for all $A \in S$, A is in the same state at the end of the executions α and β.

2.10 Shared-access and Local Time Complexity

Let *op* be a complete operation applied by p_i in execution E. The *shared-access time complexity* of operation *op* (in E) is the total number of operations that p_i performs on the base objects (equivalently, the number of steps p_i takes) in executing $\texttt{apply}(p_i, op, \mathcal{O})$ in E. The *local time complexity* of operation *op* (in E) is the total number of local computations that p_i performs in executing $\texttt{apply}(p_i, op, \mathcal{O})$ in E.

The *shared-access time complexity* of an implementation is the maximum shared-access time complexity of an operation, over all complete operations in all executions,. The *local time complexity* of an implementation is the maximum local time complexity of an operation, over all complete operations in all executions.

2.11 "Just Completes" and Solo-Termination Time Complexity

Let E be any execution fragment of $(p_1, \ldots, p_n; \mathcal{O})$, where $\mathcal{O}$ is an object implemented using a randomized implementation. We say process p_i *just completes* an operation on $\mathcal{O}$ in E if in its last step in E, p_i returns from an access procedure completing an operation on $\mathcal{O}$.

The *solo-termination shared-access time complexity of a deterministic implementation of* $\mathcal{O}$ is the maximum, over all reachable configurations C of $(p_1, \ldots, p_n; \mathcal{O})$ and all processes p_i, of $|\alpha|$ such that (1) α is a schedule that contains only p_i, and (2) in $\textsc{exec}(C, \alpha)$, p_i completes exactly one operation on $\mathcal{O}$ and, in fact, just completes it.

We note that if the solo-termination shared-access time complexity of a deterministic implementation of $\mathcal{O}$, for any deterministic solo-terminating implementation of $\mathcal{O}$, is at least k, then the shared-access time complexity of any deterministic nonblocking (or wait-free) implementation of $\mathcal{O}$ is also at least k.

Part I

Wait-Free Construction for Closed Objects

Chapter 3

Unbounded Construction for Closed Objects

3.1 Introduction

In this Chapter, we present a construction for closed objects, using registers that support LL, SC, *read* and *write* as base objects. It deviates from the final algorithm that we shall present in Chapter 5 in two ways:

1. It requires unbounded shared memory. Specifically, it requires that each process has a pool of unbounded number of cells.
2. It is not contention-sensitive: both its shared-access time complexity and its local time complexity are $O(\log^2 n)$.

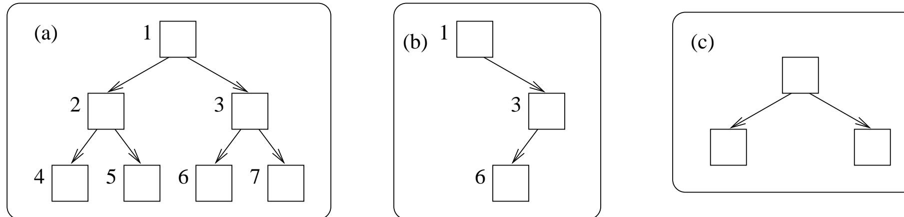
We approach our final algorithm, which requires only bounded shared memory, and is contention-sensitive, in two stages: In Chapter 4, we present an enhanced version of the unbounded algorithm in this Chapter. It requires only bounded shared memory. However, it is still not contention-sensitive. Chapter 5 presents the modification that makes our construction contention-sensitive.

3.2 Informal Description of the Construction

This section provides the intuition for how our construction works. We develop the ideas, the data structures, and informally argue the correctness of the construction.

3.2.1 Binary Tree Preliminaries

Let n be the number of processes in the system. We assume that n is a power of 2. All trees considered in this section are binary trees and are of height at most $\log n$. We say a tree is *fully formed* if its height is exactly $\log n$. We use the standard definition that a tree is *complete* if all leaves are at the same level and all interior nodes have two children. Below, we describe our scheme to number the nodes of a tree [Knu73].



(a) A fully formed tree, (b) Numbering a fully formed tree that is not complete, (c) A non-fully formed tree.

Figure 3.1: Examples of trees

First, we describe the scheme for a complete, fully formed tree. Figure 3.1(a) shows such a tree for $n = 4$. We assign the number 1 to the root and number the remaining nodes breadth-first. The number assigned to a node is its *position* in the tree. Our numbering scheme causes the leaves to be at positions $n, n+1, \ldots, 2n-1$. In particular, the Pth leaf (we consider the leftmost leaf to be the 0th leaf) is at position $n + P$. Also, the parent of a node at position i, where $i \neq 1$, is at position $\lfloor i/2 \rfloor$, and the left child and the right child of a node at position i are at positions $2i$ and $2i + 1$, respectively.

The above numbering scheme can be extended to fully formed trees that are not complete. Figure 3.1(b) presents such a tree (assuming $n = 4$). The position of a node in such a tree is the number that would be assigned to it if the tree were complete. Thus, the position of the leaf in the example tree is 6.

Finally, consider a tree that is not fully formed. For example, if $n = 4$, the tree in Figure 3.1(c) is not fully formed. In our algorithm, a non-fully-formed tree of height h will grow by getting a new root, thus becoming a sub-tree in a tree of height $h + 1$. (Thus, a leaf will always remain a leaf, as the tree grows.) Since there are multiple ways in which such a tree can be grown into a fully formed tree, it is impossible to determine the positions of the nodes without further information. However, if the position of any one node is given, the positions of others can be easily determined. For example, if (for the tree in Figure 3.1(c)) the rightmost leaf is given to be at position 5, we can infer that its parent is at position 2 and the other leaf is at position 4.

3.2.2 How Operations are Represented

We now proceed to describe the shared data structure that represents all the operations that have so far been applied on $\mathcal{O}$, and their linearization order. As a process P invokes an operation *op*, it accesses and modifies the data structure, in order to register the fact that *op* needs to be properly linearized. Once *op* has been linearized, P can compute the correct response of $\mathcal{O}$ to *op*. More than one process may compete to take steps to modify the shared data structure in incompatible ways. Our construction handles such race conditions, so that the resulting data structure remains always correct.

We now describe our data structure. A *cell* is a shared data structure with many *fields*. A field holds such information as an operation, a state of $\mathcal{O}$, or a pointer. Cells are organized into binary

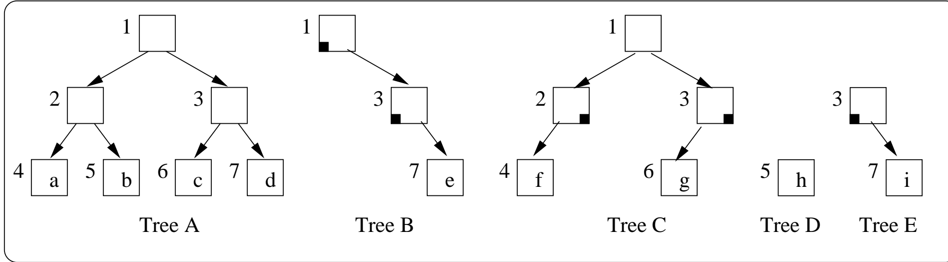


Figure 3.2: A snapshot of data structure

trees. The complete data structure consists of a sequence of fully formed trees, and a collection of non-fully formed trees.

Each process P has its own pool of unused cells. As and when P needs a new cell to install into the data structure, it creates such a cell from its pool of cells. Figure 3.2 shows an example for $n = 4$. Here Trees A, B and C are fully formed, while Trees D and E are not. The sequence of fully formed trees is: Trees A, B and C. Trees D and E are not ordered. (The number at a cell indicates its position in the tree; the letter inside a cell is the operation stored in that cell; and a black square inside a cell denotes a *nil* pointer.)

Recall that $n, n+1, \ldots, 2n-1$ are the leaf positions (in a complete, fully formed tree). In our construction, the leaf at position $n+P$ of a tree, if it exists, is associated *exclusively* with process P (processes are numbered $0, 1, \ldots, n-1$). In particular, if a tree has a cell at position $n+P$, that cell was introduced into the data structure by process P and it stores an operation from process P. Thus, Tree A (in Figure 3.2) contains operations from all four processes — operation a from Process 0 (at position 4), operation b from Process 1, and operations c and d from processes 2 and 3, respectively. Tree B contains only one operation — operation e from Process 3 (looking at the shape of this tree, we know that the only leaf is at position 7 and hence contains an operation from Process 3). Tree C contains two operations — operation f from Process 0 and operation g from Process 2. Tree D is not fully formed (since its height is less than $\log n$). In fact, given that this tree has only one cell so far, it would be normally impossible to say what the position of this cell is. Our construction, however, ensures that each tree grows bottom-up — from leaves to the root. Thus, one can conclude that the lone cell in Tree D will eventually be a leaf (in a fully formed tree). As already mentioned, our construction also ensures that a leaf created by process P will be at position $n+P$ of a fully formed tree. Thus, assuming that the cell in Tree D was created by Process 1, we indicated its position as 5 (in a fully formed tree that this cell will eventually be a part of). Tree E in the figure is also not fully formed. It contains one operation — operation i from Process 3.

3.2.3 Linearization Order

In our construction, an operation takes effect the moment it becomes a part of a fully formed tree. Thus, in our example in Figure 3.2, operations a through g took effect, but operations h and i did not. All fully formed trees form a total order Ω, according to the times their cells at position 1 are formed: at the moment a tree becomes fully formed by getting a cell at position 1, it becomes

the last tree in Ω. Operations that took effect are linearized according to the following rules: if Tree A precedes Tree B in Ω, then all operations in Tree A are linearized before any operation in Tree B. Operations within a tree are linearized in the natural left to right order on the leaves. Thus, in our example, a, b, c, d, e, f, g is the linearization order. Since the two non-fully-formed trees (Trees D and E) can grow to become fully formed trees in many ways, it is possible that operation i may precede h, and be separated from h by other operations, in the final linearization order.

3.2.4 Growing Trees Bottom-up

In our construction, when a process P wishes to apply an operation *op* on the implemented object, P gets a cell from its private pool of cells, stores *op* in the cell, and creates a tree out of this single cell. (Tree D in Figure 3.2 is an example of such a tree: here Process 1 has just created a tree out of its operation cell.) P then makes an effort to grow this (single-celled) tree into a fully formed tree (of height $\log n$) in which its operation cell will be a leaf at position $n + P$. We already mentioned that P's operation takes effect only after its operation cell becomes a leaf of a fully formed tree. Since a process returns from its operation only after the operation takes effect, our construction satisfies the following property:

> **Completion Rule**: A process P returns from its operation only after its operation cell is a leaf (at position $n + P$) of a fully formed tree.

Our construction also ensures that a tree always grows bottom-up. Specifically, the construction satisfies the following property:

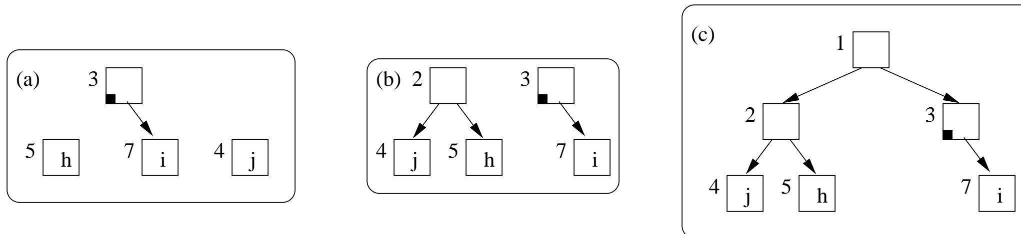
> **Bottom-up Rule**: If C is a cell in a tree, then the subtree rooted at C does not subsequently change.

For example, consider Tree E in Figure 3.2. The parent of the cell containing operation i has no left child. By the bottom-up rule, this cell will never get a left child in the future.

It is immediate from the bottom-up rule that a fully formed tree never subsequently changes: there won't be any addition or removal of cells from a fully formed tree. This fact implies that our linearization order respects operation precedence, as we argue now. Consider two operations *op* and *op*$'$ such that *op* precedes *op*$'$ (*i.e.*, *op* returns before *op*$'$ is invoked). By the completion rule, when *op*$'$ is invoked, *op* is already a leaf of a fully formed tree. Since fully formed trees never subsequently change, *op*$'$ becomes a leaf in a later tree. As a result, by our linearization rule, *op* is linearized before *op*$'$, as required.

3.2.5 How a Process Makes Progress

To understand how processes make progress, let us consider a specific example: suppose that Process 0 invokes the operation j and runs alone from the configuration depicted in Figure 3.2. (In Figures 3.3(a)-3.3(c), we will depict the sequence of changes to the configuration. Since the three fully formed trees are a part of all of these configurations, we will not repeat them in these figures.) Process 0 creates a (single-celled) tree containing its operation cell (Figure 3.3(a)). The



(a) Process 0 creates an operation cell for j, (b) Process 0 creates a cell whose left and right children are j and h,
(c) Process 0 creates a cell at position 1. j has now taken effect.

Figure 3.3: Example of a process making progress

next goal of Process 0 is to make this operation cell a leaf (at position $n+P = n$) of a fully formed tree; *i.e.*, to get this cell a parent (at position $\lfloor (n+P)/2 \rfloor = n/2$), then a grandparent (at position $\lfloor (n+P)/4 \rfloor = n/4$), then a parent for the grandparent and so on, until its operation cell has $\log n$ ancestors. Thus, the immediate subgoal is to get a parent for its operation cell (containing the operation j). Since the parent will be a cell at position $\lfloor (n+P)/2 \rfloor$, which in our example is 2, the parent can potentially have two children: one at position 4 and another at position 5. Therefore, Process 0 checks if there is already an unparented cell at position 5 and finds the operation cell containing h. So it creates a cell whose left and right children are the operation cells containing j and h, respectively (Figure 3.3(b)). Process 0 then proceeds to get a parent (at position 1) for this new cell. Since a cell at position 1 can potentially have cells at positions 2 and 3 as its left and right children, it checks if there is an unparented cell at position 3. Since there is such a cell, it creates a cell at position 1 as shown in Figure 3.3(c). At this point, the operation j of Process 0 has taken effect. So have the operations h and i of processes 1 and 3: the steps of Process 0 have helped them take effect as well.

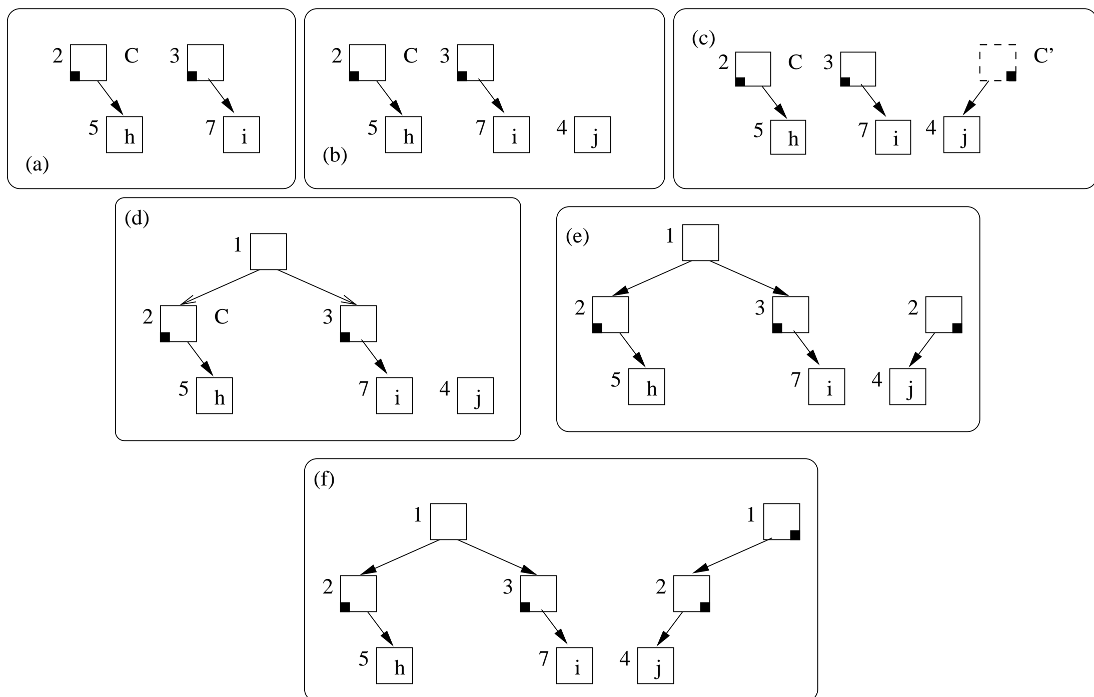
The above example also roughly suggests why our shared-access and local time complexity has a logarithmic, rather than linear, dependence on n.

To understand certain intricacies of our construction, let us consider another example: a run, again starting from the configuration in Figure 3.2, in which the steps of Process 0 and Process 1 interleave. (We will depict the sequence of changes to the configuration in Figures 3.4(a)-3.4(f).) Specifically, the run is as follows:

- Process 1, which has already announced its operation h, proceeds to get a parent for its operation cell. This parent can potentially have a cell at position 4 as its left child; however, Process 1 does not find an unparented cell at position 4. So it creates a cell C whose left child is *nil* and right child is the operation cell containing h (see Figure 3.4(a)).

- Process 0 announces a new operation j (Figure 3.4(b)).

- Process 0 proceeds to get a parent for its operation cell. Since this parent can potentially have a cell at position 5 as its right child, it looks for an unparented cell at position 5. Since there is no such cell, it creates a cell C' whose right child is *nil* and left child is its operation cell (see Figure 3.4(c)).



(a) Process 1 creates a parent for h, (b) Process 0 creates an operation cell for j,

(c) Process 0's attempt to install C' into global structure fails, (d) Since process 0 cannot create a parent for j, it gets a parent for C,

(e) Process 0 gets a parent for j, (f) Process 0 gets a grandparent for j. Operation j has now taken effect.

Figure 3.4: Another example of a process making progress

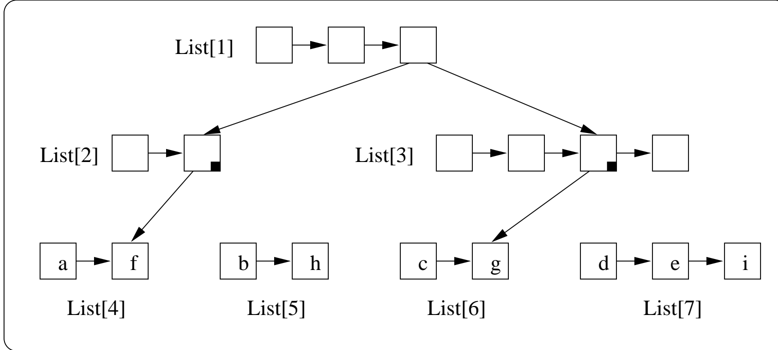


Figure 3.5: The lists for the trees in Figure 3.2

Notice that Cell C', like Cell C, is a cell at position 2. If we make it possible for Process 0 to install cell C' into the data structure, there will be two unparented cells at position 2. Our construction does not permit this. Specifically, our construction satisfies the following invariant:

> **At Most One Orphan Rule**: There is at most one unparented cell in any non-root position at any time.

Accordingly, the attempt of Process 0 to install C' into the data structure (at position 2) will fail.

- Suppose Process 0 is the only process alive. Clearly, because of "At most one orphan rule" and "bottom-up rule" (described earlier), Process 0 will not be able to get a parent for its operation cell until C gets a parent. So Process 0 gets a parent for C, even though its operation cell is not a descendant of C (see Figure 3.4(d)).
- Process 0 then gets a parent for its operation cell (Figure 3.4(e)) and then a grandparent (Figure 3.4(f)). At this point, its operation j has taken effect.

3.3 The Data Structure

We present our unbounded construction for closed objects in Figures 3.6 to 3.9. In this section, we give a concrete description of the data structure used by our algorithm, and state the rules that our algorithm respects when manipulating the data structure.

As already explained, the principal data structure maintained by our algorithm is a sequence of trees. With the passage of time, this sequence grows without bound. The sequence of trees at a point in a run is represented using $2n-1$ singly linked lists (we will use the sequence of trees in Figure 3.2 as our running example). The lists are denoted by $List[1], List[2], \cdots, List[2n-1]$. $List[lst]$ consists of all cells at position lst in the sequence of trees. Figure 3.5 depicts the lists for the trees in Figure 3.2. The parent-child relationship within each tree is preserved by including appropriate left child and right child pointers between cells in different lists. For example, Figure

3.5 depicts these pointers corresponding to Tree C in Figure 3.2. (To keep the figure uncluttered, the left child and right child pointers corresponding to the remaining trees are not shown.)

We call $List[1]$ the *root list*, $List[lst]$, where lst is between n and $2n-1$, a *leaf list*, and $List[lst]$, where lst is between 2 and $n-1$, a *non-root interior list*. A cell in the root list is called a *root cell*. A cell in a leaf list is called a *leaf cell*. A cell in a non-root interior list is called a *non-root interior cell*.

Notice that, if c is a cell in $List[lst]$, either c has no left child or c's left child is a cell in $List[2 * lst]$. Similarly, either c has no right child or c's right child is a cell in $List[2 * lst + 1]$.

3.3.1 The Representation of a List

3.3.1.1 The Fields of a Cell

The various fields of a cell in a list are as follows:

- The *LC*, *RC*, and *Parent* fields of a cell point to the left child, right child, and parent of that cell. (If a cell has no left child, right child or parent, the corresponding field has $\perp$).

- The *Next* field points to the next cell in the same list. (If a cell is the last one in a list, its *Next* field has $\perp$).

- The *Op* field of a cell y contains a single operation that results from combining the operations at the leaves of the subtree rooted at y. For example, the *Op* field of the first cell in List[1] contains the operation $a \bigotimes b \bigotimes c \bigotimes d$. The *Op* field of the first cell in List[3] contains $c \bigotimes d$, and the *Op* field of the first cell in List[4] contains a.

- The *Lop* field of a cell y contains a single operation that results from combining the operations at the leaves of the left subtree of y (if y has no left child, then y's *Lop* field has the value $\perp$). For example, the *Lop* field of the first cell in List[1] contains $a \bigotimes b$, the *Lop* field of the first cell in List[3] contains c, and the *Lop* field of the first cell in List[4] contains $\perp$.

- The *State* field of a cell y is used for recording the state that results from applying all operations at the leaves to the left of y, in the natural left-to-right order, to the implemented object in the initial state.[1] (Leaves to the left of y are all the leaves, in all trees, that are to the left of the leftmost leaf descendant of y.) For example, the *State* field of the third cell in List[3] contains the state that results from applying the operations a, b, c, d, e, f, in that order.

- Finally, each cell has a *Ready* field that holds a boolean value. A value of *false* indicates that the cell is not yet ready to have a parent, while a value of *true* indicates that the cell is ready to have a parent. As will be clear later, this field helps the algorithm ensure that a cell becomes a "full-fledged" member of its list before its parent becomes a full-fledged member of the parent-list.

[1]The initial state of the implemented object is specified in the implementation.

3.3.1.2 The *Head* Variables

Associated with each $List[lst]$, $1 \leq lst \leq 2n-1$, there is a shared variable $Head[lst]$ that intends to point to the last cell in that list.[2] Since a cell c is first appended to $List[lst]$ and only later is $Head[lst]$ updated to point to c, there is a small window of time during which $Head[lst]$ points not to the last cell in the list, but to the cell previous to the last.

Initially, each $List[lst]$ has a single dummy cell, that we call $anchor_{lst}$, and $Head[lst]$ points to this cell. (See Lines 1-4 of the `initialization` section of the algorithm in Figure 3.6.)

3.3.2 Stability and Order Properties

Our algorithm ensures that the fields of a cell are stable: once a field is assigned a non-$\perp$ value, it remains unchanged forever. The algorithm also ensures that events—modifying a field, appending a cell to a list, modifying a *Head* variable—occur in a particular order. The stability and the order properties, which we state below, are crucial to the correctness of the algorithm.

Let c be a cell in $List[lst]$, $1 \leq lst \leq 2n-1$. The values in the following fields of c remain unchanged from the time when c first becomes a cell in $List[lst]$: $c \rightarrow LC$, $c \rightarrow RC$, $c \rightarrow Op$, $c \rightarrow Lop$.

The order in which the remaining fields of a cell are updated is slightly different depending on whether the cell is in the root list (*i.e.*, $List[1]$), a leaf list, or a non-root interior list. Below, we therefore consider these cases separately.

3.3.2.1 Stability and Order Properties for a Non-root Interior Cell

Order of Events

Let d be a cell (other than the first cell) in $List[lst]$, $2 \leq lst \leq n-1$, c be d's immediate predecessor (*i.e.*, $c \rightarrow Next$ points to d), and e be d's parent (in $List[\lfloor lst/2 \rfloor]$). Then, the appending of d to $List[lst]$, the appending of e to $List[\lfloor lst/2 \rfloor]$, and the processing of d's fields must have occurred in the following order:

1. $Head[lst]$ points to c, and c is the last cell in $List[lst]$.
2. d is appended to $List[lst]$, *i.e.*, $c \rightarrow Next$ gets the value d.
3. d becomes ready, *i.e.*, $d \rightarrow Ready$ is assigned *true*.
4. e is appended to $List[\lfloor lst/2 \rfloor]$. At this point, $e \rightarrow LC$ or $e \rightarrow RC$, whichever is appropriate, holds the value d.
5. $d \rightarrow Parent$ is assigned the value e.
6. $Head[lst]$ is updated to point to d, and at this point d is the last cell in $List[lst]$.
7. e becomes ready.

[2]Even though this variable points to the tail of a list, it is called *Head* to maintain consistency with established usage.

8. $d \rightarrow State$ is assigned a non-$\perp$ value.

The above order is presented here because it is crucial to our proof of correctness. (Where no explicit ordering between the updating of two fields in the data structure is implied by the above order, there may or may not be an ordering constraint. Our formal proof will reveal all such ordering constraints.)

Stability of fields

Consider the point t in time when a cell d is appended to a cell c in $List[lst]$, $2 \leq lst \leq n-1$ (*i.e.*, at time t, $c \rightarrow Next$ is assigned d). The stability properties are stated with respect to time t as follows:

(S1.) The values of $d \rightarrow LC$ and $d \rightarrow RC$ will never subsequently change (from their values when d is appended to the list).

This property captures the fact that trees grow bottom-up: From the time that d is appended to the list, no nodes will be added to, or removed from the subtree rooted at d (this subtree consists of d, its children, their children and so on).

(S2.) When d is appended to the list, $d \rightarrow Op$ holds the operation that combines the *Op* fields of the left child and the right child of d. The value of $d \rightarrow Op$ will never subsequently change.

It follows that that $d \rightarrow Op$ always holds the operation obtained by combining the operations at the leaf descendants of d.

(S3.) When d is appended to the list, $d \rightarrow Lop$ holds the operation in the *Op* field of the left child of d. The value of $d \rightarrow Lop$ will never subsequently change.

It follows that $d \rightarrow Lop$ always holds the operation obtained by combining the operations at the leaf descendants of the left child of d.

(S4.) When d is appended to the list, $d \rightarrow Parent$ has $\perp$. The value of this field changes at most once in the future.

(S5.) When d is appended to the list, $d \rightarrow Next$ has $\perp$. The value of this field changes at most once in the future.

(S6.) When d is appended to the list, $d \rightarrow Ready$ has *false*. The value of this field changes at most once in the future (to *true*).

(S7.) When d is appended to the list, $d \rightarrow State$ has $\perp$. The value of this field changes at most once in the future.

3.3.2.2 Stability and Order Properties for a Leaf Cell

If d is a cell in a leaf list, *i.e.*, in $List[lst]$, where lst is between n and $2n-1$, then the stability and order properties are the same as above with one change: items (2) and (3) are interchanged in the order. Thus, when d is appended, $d \rightarrow Ready$ has *true* and this value will never subsequently change.

3.3.2.3 Stability and Order Properties for a Root Cell

Let d be a cell in the root list and, as before, let c be d's immediate predecessor in List[1] (in this case, there is no e, the parent of d, because a cell in the root list does not have a parent). The stability properties of the fields of d are the same as before (except that $d \rightarrow Parent$ remains unchanged from its value of $\perp$). The order of events is now: items (1), (2), (3), (8), (6). Thus, $d \rightarrow State$ is assigned a non-$\perp$ value *before* $Head[1]$ is updated to point to d.

3.3.3 Facts about the $Head$ Variables

A cell d in $List[lst]$ *has a parent* if there is a cell e in $List[\lfloor lst/2 \rfloor]$ such that $e \rightarrow LC$ or $e \rightarrow RC$ has the value d. Notice that during the interval lasting from when e is appended to $List[\lfloor lst/2 \rfloor]$ to when d's *Parent* field is assigned e, d's parent field has $\perp$ even though, by our definition, d has a parent. A cell is an *orphan* if it has no parent. We say a cell has a *ready parent* if it has a parent whose ready field has *true*. Similarly, a *ready orphan* is an orphan whose ready field has *true*.

The following property states certain facts about the *Head* variables. We don't prove it here, but it follows easily from the order of events stated above. (This order of events is enforced by our algorithm.)

(P1.) This property is stated in three parts.

1. For any $List[lst]$, $1 \leq lst \leq 2n-1$, we have:
   (a) $Head[lst]$ is never $\perp$, and whenever the value of $Head[lst]$ changes from c to d, we have $c \rightarrow Next = d$.
   (b) $Head[lst]$ points either to the last cell or the cell immediately before the last cell in $List[lst]$.
2. For any non-root list $List[lst]$, $2 \leq lst \leq 2n-1$, we have:
   (a) If $Head[lst]$ points to a cell c, then c and every cell before c in $List[lst]$ has a parent in $List[\lfloor lst/2 \rfloor]$.
   (b) If a cell c in $List[lst]$ has a ready parent, then $Head[lst]$ points to c or a cell after c in $List[lst]$.
3. For the root list (*i.e.*, $List[1]$), the following holds: If $Head[1]$ points to a cell c, then the state field of c and the state field of every cell before c in List[1] has a non-$\perp$ value.

3.3.4 Definition of Correct State

For any cell c in a fully formed tree, the *operation sequence preceding* c is is the sequence of operations at the leaves (in that tree and all preceding trees) that are to the left of the left-most leaf descendant of c. For example, for the cell at position 3 of Tree C of Figure 3.2, this sequence is a, b, c, d, e, f. As another example, for the cell at position 3 of Tree A, this sequence is a, b.

For a cell c in a fully formed tree, let $op_1, op_2, \ldots, op_k$ denote the operation sequence preceding c. We define the *correct state for cell* c as the state that results from applying the operations $op_1, op_2, \ldots, op_k$, in that order. More precisely, the correct state for c is $\delta_{state}(\text{INITIALSTATE}, op)$, where $op = op_1 \bigotimes op_2 \bigotimes \cdots \bigotimes op_k$.

A crucial aspect of the proof of correctness of our algorithm is to show that, for each cell c, the algorithm writes into c's state field the correct state for c.

3.4 How the Algorithm Works

In this section, we describe how the various procedures constituting the algorithm work together. We argue the correctness of each procedure by top-down reasoning: if procedure A calls procedure B, we argue that A satisfies its stated properties on the premise that B satisfies its properties, and later justify the premise. The presentation is informal and intended to provide an intuitive understanding of the algorithm. A rigorous proof of correctness is provided in the next section.

3.4.1 How `apply` Works

A process P applies an operation *op* on the implemented object by executing the procedure `apply`$(P, op, \mathcal{O})$. We argue the correctness of the response returned by `apply` on the premise that `promote` and `percolateState` satisfy certain properties, stated informally as follows:

(P2.) If cell c in a non-root list $List[lst]$ is ready, then an execution of `promote`(lst) ensures that c has a ready parent.

(P3.) If c is a ready cell in the root list that has $\perp$ in its state field, an execution of `promote`(1) ensures that (1) c's state field holds the correct state for c, and (2) $Head[1]$ points to c or beyond.

(P4.) If c is a cell in $List[lst]$ and the state field of c's root-ancestor has the correct state, then an execution of `percolateState`(c, lst) ensures that the correct state for c is written into c's state field.

When a process P calls `apply`$(P, op, \mathcal{O})$, it first executes `announce`(op, P). During this procedure, P grabs a fresh cell, *opcell*, from its private pool of cells, stores *op* in the operation field, marks the cell ready, and appends *opcell* to the end of P's leaf list, namely, $List[n + P]$. [3] P then calls `promote`$(n + P)$ which, by Property P2 stated above, results in *opcell* getting a ready parent. P then calls `promote`$(\lfloor (n+P)/2 \rfloor)$, then `promote`$(\lfloor (n+P)/4 \rfloor)$, and so on, so that *opcell* gets a ready grandparent, a ready parent to grandparent, and so on. Ultimately, after $\log n$ such calls, *opcell* has a ready ancestor c in the root list, *i.e.*, $List[1]$. P then calls `promote`(1) which, by Property P3, ensures that c's state field holds the correct state for c. P then calls `percolateState` which, by Property P4, ensures that the state field of *opcell* holds the correct state. P applies its operation *op* to this state, and returns the resulting response as the implementation's response to *op*.

[3] The use of LL,SC in Lines 3,4 of `announce`(op, P) can in fact be replaced with *write*. However, the use of LL, SC here allows for a cleaner proof of correctness.

```
initialization
∀i, 1 ≤ i ≤ 2n-1 : anchor_i : *CELL

1.  for i := 1 to 2n-1:
2.     anchor_i → Next := ⊥, Head[i] := anchor_i
3.  anchor_1 → State := INITIALSTATE
4.  anchor_1 → Op := ⊥
end initialization


apply (P : INTEGER, op : OP, 𝒪) returns RES
1.  opcell := announce (op, P)
2.  for i := 0 to log n
3.     promote (⌊(n+P)/2^i⌋)
4.  percolateState (opcell, n+P)
5.  return δ_resp (opcell → State, op)
end apply


promote (lst : INTEGER)
1.  if lst = 1
2.     head := LL(Head[lst])
3.     newcell := head → Next
4.     if newcell = ⊥ return
5.     if newcell → Ready = false return
6.     newcell → State := δ_state (head → State, head → Op)
7.     SC(Head[lst], newcell)
8.     return
9.  append (⌊lst/2⌋)
10. promote (⌊lst/2⌋)
11. append (⌊lst/2⌋)
end promote
```

Figure 3.6: Unbounded construction for closed object $\mathcal{O}$ (Figures 3.6 to 3.9)

```
    append (lst : INTEGER)
1.    head := Head[lst]
2.    newcell := LL(head → Next)
3.    if newcell = ⊥
4.       (lc, lop) := readyOrphan (2 ∗ lst)
5.       (rc, rop) := readyOrphan (2 ∗ lst + 1)
6.       if (lc ≠ ⊥) or (rc ≠ ⊥)
7.          c := combine (lc, lop, rc, rop)
8.          SC (head → Next, c)
9.       newcell := head → Next
10.      if newcell = ⊥ return

11.   lchild := newcell → LC
12.   if lchild ≠ ⊥
13.      if LL(lchild → Parent) = ⊥
14.         SC(lchild → Parent, newcell)
15.      lchead := LL(Head[2*lst])
16.      if lchead → Next = lchild
17.         SC(Head[2*lst], lchild)

18.   rchild := newcell → RC
19.   if rchild ≠ ⊥
20.      if LL(rchild → Parent) = ⊥
21.         SC(rchild → Parent, newcell)
22.      rchead := LL(Head[2*lst+1])
23.      if rchead → Next = rchild
24.         SC(Head[2*lst+1], rchild)

25.   newcell → Ready := true
    end append
```

Figure 3.7: Unbounded construction for closed object $\mathcal{O}$ (Figures 3.6 to 3.9)

```
readyOrphan (lst : INTEGER) returns (*CELL, OP)
1.  head := Head[lst]
2.  newcell := head → Next
3.  if newcell = ⊥  return (⊥, ⊥)
4.  if newcell → Ready = false return (⊥, ⊥)
5.  return (newcell, newcell → Op)
end readyOrphan
```

```
announce (op : OP, P : INTEGER) returns *CELL
1.  allocate a new cell c and initialize it as follows:
        c → Parent := ⊥, c → Next := ⊥, c → State := ⊥
        c → LC := ⊥, c → RC := ⊥,
        c → Op := op, c → Lop := ⊥, c → Ready := true
2.  head := Head[n+P]
3.  if LL(head → Next) = ⊥
4.      SC(head → Next, c)
5.  return c
end announce
```

Figure 3.8: Unbounded construction for closed object $\mathcal{O}$ (Figures 3.6 to 3.9)

```
   combine (lc : *CELL, lop : OP, rc : *CELL, rop : OP) returns *CELL
1.    allocate a new cell c and initialize it as follows:
         c → Parent := ⊥, c → Next := ⊥, c → State := ⊥
         c → LC := lc, c → RC := rc,
         c → Lop := lop, c → Op := (lop ⊗ rop)
         c → Ready := false
2.    return c
   end combine
```

```
   percolateState (c : *CELL, lst : INTEGER)
1.    if lst = 1 return
2.    p := c → Parent
3.    percolateState (p, ⌊lst/2⌋)
4.    if lst = 2 * ⌊lst/2⌋
5.       c → State := p → State
6.    else c → State := δ_state (p → State, p → Lop)
   end percolateState
```

Figure 3.9: Unbounded construction for closed object $\mathcal{O}$ (Figures 3.6 to 3.9)

3.4.2 How `promote` Works on a Non-root List

We now argue that `promote`(lst), $lst > 1$, satisfies Property P2 on the premise that the following two properties are true.

(P5.) At the time that a cell c in $List[lst]$ becomes a ready orphan, suppose that the head of the parent list (*i.e.*, $Head[\lfloor lst/2 \rfloor]$) points to cell d. Then, the parent that c will get in the future will be within two cells from d (*i.e.*, if e and f are the cells that will immediately follow d, then one of e and f will be c's parent).

(P6.) At a time that a cell c in $List[lst]$ is a ready orphan, suppose that $Head[\lfloor lst/2 \rfloor]$ points to a cell d. Then, an execution of `append`($\lfloor lst/2 \rfloor$) ensures that d has a ready cell immediately following it.

To verify that `promote` satisfies Property P2, suppose that cell c in a non-root list $List[lst]$ is ready before an execution of `promote`(lst). Our aim is to show that after `promote`(lst) terminates, c has a ready parent. Let t be the time when c first becomes a ready orphan. Let $Head[\lfloor lst/2 \rfloor]$ point to cell d at time t. Clearly, c's parent cannot be d or any cell preceding d in $List[\lfloor lst/2 \rfloor]$.

The execution of `promote`(lst) begins with `append`($\lfloor lst/2 \rfloor$) (on Line 9 of `promote`). By Property P6, when `append` terminates, d has a ready cell e immediately following it. If e is c's parent, then we have Property P2.

Suppose that e is not c's parent. After the recursive call to `promote`($\lfloor lst/2 \rfloor$) on Line 10, $Head[\lfloor lst/2 \rfloor]$ points to e or beyond (we denote this fact by (@)), as argued below. If $\lfloor lst/2 \rfloor > 1$, Line 10 ensures, by an inductive application of Property P2, that e has a ready parent; then, by Property P1(2b), $Head[\lfloor lst/2 \rfloor]$ points to e or beyond. If $\lfloor lst/2 \rfloor = 1$, the recursive call on Line 10 ensures, by Property P3, that $Head[\lfloor lst/2 \rfloor]$ points to e or beyond.

Let t' be the time when $Head[\lfloor lst/2 \rfloor]$ first points to e. By the order of events, e is the last cell in $List[\lfloor lst/2 \rfloor]$ at t'. We note that t' is a time before the execution of `append`($\lfloor lst/2 \rfloor$) on Line 11 of `promote` (by (@)), and that at t', c in $List[lst]$ is a ready orphan (because e is not c's parent), $Head[\lfloor lst/2 \rfloor]$ points to cell e. We now apply Property P6 to this execution of `append`($\lfloor lst/2 \rfloor$). Thus, when `append`($\lfloor lst/2 \rfloor$) on Line 11 terminates, e has a ready cell f immediately following it. By Property P5, f is c's parent and so, we have Property P2.

3.4.3 How `promote` Works on a Root List

To verify that `promote`(1) satisfies Property P3, suppose that c is a ready cell in List[1] that has $\perp$ in its state field. By the conjunction of Property P1(1b) and P1(3), $Head[1]$ points to the cell b immediately before c. Thus, when `promote`(1) is executed, *head* is assigned b (Line 2), *newcell* is assigned c (Line 3), and *newcell* is found to be non-$\perp$ and ready (Lines 4 and 5). Then, b's state field holds a non-$\perp$ value s (by Property P1(3)) and b's operation field combines the operations at the leaves of b. Assuming that s is correct for b, it is obvious that $s' = \delta_{state}(s, b \rightarrow Op)$ is correct for c. Line 6 writes s' into c's state field. If the SC on Line 7 succeeds, $Head[1]$ points to c; otherwise some other process must have already updated $Head[1]$ to c. Hence, we have Property P3.

3.4.4 How append Works

A leaf list, *i.e.*, $List[n+P]$ $(0 \leq P \leq n-1)$, grows when process P executes `announce`(op, P) and appends its operation cell to the list. An interior list $List[lst]$ $(1 \leq lst \leq n-1)$, on the other hand, grows when a process P executes `append`(lst). Roughly speaking, this procedure finds the "end" of $List[lst]$ and appends there a cell that becomes the parent to ready orphans in $List[2*lst]$ and $List[2*lst+1]$.

We now describe informally how process P executes `append`(lst). P reads $Head[lst]$ and obtains a pointer to either the last or the previous-to-last cell in $List[lst]$ (Line 1). To distinguish between the two cases, P inspects the next field of *head* (Line 2). If there is a cell next to *head*, P proceeds to help it (Line 11 onwards). Otherwise P looks at the child lists, namely, $List[2*lst]$ and $List[2*lst+1]$, to check if they have ready orphans (Lines 4 and 5). If so, P attempts to create a parent for these ready orphans by combining the information in the ready orphans in a new cell (Lines 6 and 7), and then attempting to append the new cell to the end of the $List[lst]$ (Line 8). Its attempt may fail if some other process has already appended a cell to *head*. Regardless of whether it succeeded in appending *newcell* or someone else succeeded in appending some other cell, P obtains a pointer to the cell next to *head* (Line 9). If no cell exists there, it means that there was nothing to append to *head*, and so P returns from the procedure (Line 10). Otherwise, *newcell* is a cell next to *head*, and P proceeds to notify *newcell*'s children of the fact that they now have a parent. It sets the parent field of *newcell*'s left child to *newcell* (Lines 12-14) and, since *newcell*'s left child is now parented, updates $Head[2*lst]$ to point to $newcell \rightarrow LC$ (Line 15-17). P notifies *newcell*'s right child similarly (Lines 18-24). Once *newcell*'s children have been notified, *newcell* is allowed to have a parent. To reflect this fact, P sets *newcell*'s ready field to *true* (Line 25).

Properties P5 and P6 follow easily from the way the `append` procedure is designed. Their proofs, however, are long and tedious because of the need to address many cases. So we defer rigorous proofs to the next section and, in the following, only informally describe why the properties hold.

3.4.4.1 Why Property P5 Holds

We now explain why Property P5 holds. Suppose that $Head[\lfloor lst/2 \rfloor]$ points to a cell d at the time t when a cell c becomes a ready orphan in $List[lst]$. Let b be the cell whose next field points to c. By the conjunction of Property P1(1b) and P1(2a), $Head[lst]$ points to b at time t. Let e and f be the two cells that immediately follow d, at some instant of time after t. To verify P5, we show that either e or f is c's parent. (Clearly, c's parent cannot be d or any cell preceding d.) It suffices to show that, if e is not c's parent, then f is.

Suppose that e is not c's parent. We claim that at any time after t, and before f is appended next to e, $Head[lst]$ points to b (We denote this claim by (*)). This claim is true for the following reason: By the order of events, if $Head[lst]$ points beyond b, then c already has a parent. However, by assumption, c does not have a parent before f is appended next to e. Thus, claim (*) holds.

Consider the execution $\mathcal{A}$ of `append`$(\lfloor lst/2 \rfloor)$ that will append f next to e. $\mathcal{A}$ must have obtained e when it read $Head[\lfloor lst/2 \rfloor]$ on Line 1. This implies that $\mathcal{A}$ performed Line 1 *after* time t (because, at time t, $Head[\lfloor lst/2 \rfloor]$ was pointing to d, not e). By claim (*), when Lines 4 and 5

are performed by $\mathcal{A}$, the ready orphan c in $List[lst]$ will be noticed by `combine` (called in Line 7 of `append`), and will become cell f's child. Thus, cell f appended next to e will be c's parent. Hence, we have Property P5.

3.4.4.2 Why Property P6 Holds

To understand why Property P6 holds, suppose that at time t $List[lst]$ has a ready orphan c and $Head[\lfloor lst/2 \rfloor]$ points to d. Consider an execution $\mathcal{A}$ of `append`($\lfloor lst/2 \rfloor$) that begins after time t. When $\mathcal{A}$ reads $Head[\lfloor lst/2 \rfloor]$ on Line 1, there are three possibilities for what it obtains: (1) it obtains d, (2) it obtains a cell e immediately following d, or (3) it obtains a cell f that follows d, but not immediately following d. Let us first consider the last two cases. By Property P1(2a), the cell that $Head[\lfloor lst/2 \rfloor]$ points to and every preceding cell has a parent. Further, by the order of events, a non-root cell must be ready before it has a parent. Therefore, in Case (3), f and every cell preceding f has a parent and is ready. In particular, it follows that the cell immediately following d is ready. Similarly, in Case (2), the cell e (which immediately follows d) has a parent and is ready. This establishes Property P6 for cases (2) and (3).

Let us now consider Case (1): when $\mathcal{A}$ reads $Head[\lfloor lst/2 \rfloor]$ on Line 1, it obtains d (so *head* gets the value d). Then, when $\mathcal{A}$ performs LL($d \rightarrow Next$) on Line 2, there are two sub-cases: (a) it obtains $\perp$, or (b) it obtains a non-$\perp$ pointer e. In Subcase (b), when $\mathcal{A}$ eventually executes Line 25, $e \rightarrow Ready$ is set to *true*, thus satisfying Property P6.

In Subcase (a), $\mathcal{A}$ proceeds to execute Lines 4-10. When $\mathcal{A}$ executes Lines 4 and 5, c may or may not be an orphan. Consider the case that $\mathcal{A}$ finds c to be a ready orphan in $List[lst]$. In this case, the if-condition on Line 6 holds true, and so $\mathcal{A}$ executes Lines 7 and 8. If the SC on Line 8 succeeds, the reading of $d \rightarrow Next$ (Line 9) clearly returns a non-$\perp$ e and, subsequently, on Line 25 e's ready field is set to *true*, thus satisfying P6. If the SC on Line 8 fails, then some other process Q must have performed a successful SC, between the times when $\mathcal{A}$ executed Lines 2 and 8, that appended a cell e next to d. On Line 9, $\mathcal{A}$'s reading of $d \rightarrow Next$ returns that cell e and, subsequently, on Line 25 e's ready field is set to *true*, thus satisfying P6. This leaves just one case to consider: on Lines 4 and 5, $\mathcal{A}$ does not find c to be a ready orphan in $List[lst]$. This case implies that, when $\mathcal{A}$ performs Lines 4 and 5, $Head[lst]$ already points to c or beyond (otherwise $\mathcal{A}$ would find c to be a ready orphan). It follows, by Property P1(2a), that c has a parent p (in $List[\lfloor lst/2 \rfloor]$). Since c was an orphan at time t (when $Head[\lfloor lst/2 \rfloor]$ pointed to d), it follows that p is after d. In particular, it follows that $d \rightarrow Next$ is non-$\perp$. This implies that, when $\mathcal{A}$ reads $d \rightarrow Next$ on Line 9, it obtains a non-$\perp$ cell e. Subsequently, on Line 25, $\mathcal{A}$ sets e's ready field to *true*, thus satisfying P6.

3.4.5 How `percolateState` Works

To verify that `percolateState` satisfies Property P4, let c be a cell in $List[lst]$ and suppose that the state field of c's root-ancestor r holds the correct state. We argue that after executing `percolateState`(c, lst), c holds the correct state. If $lst = 1$, we have $c = r$. Thus, P4 trivially holds. Otherwise let d be c's parent (Line 2). By induction, the recursive call on Line 3 ensures that d holds the correct state. If c is the left child of d, then by the definition of correct state, the correct state for c is the same as for d. This observation justifies Lines 4 and 5. If c is the right

child of d, then by the definition of correct state, the correct state for c is obtained by applying the operations in the left subtree of d to the state in d. This is done in Line 5.

3.5 Proof of Correctness

We present now a proof of correctness of our closed object construction. In Section 3.5.1, we prove that when `apply`$(P, op, \mathcal{O})$ completes, the operation cell that represents *op* has a root ancestor in the sequence of trees. Thus, *op* has indeed taken effect. In Section 3.5.2, we prove that the value returned by `apply`$(P, op, \mathcal{O})$ is consistent with the intended linearization order on the operations, as represented by the sequence of trees.

3.5.1 Proof of Progress

3.5.1.1 Reachable Cell

We first show that the fields of a cell are *stable*, *i.e.* once such a field holds a non-trivial value, that value never changes. More specifically, for any cell c, $c \rightarrow LC$, $c \rightarrow RC$, $c \rightarrow Lop$, and $c \rightarrow Op$ are unchanged throughout a run. Further, once $c \rightarrow Next$ and $c \rightarrow Parent$ hold a non-$\perp$ value d, then they hold the value d forever. Finally, once $c \rightarrow Ready$ becomes *true*, it stays *true* forever. (The stability of $c \rightarrow State$ is proved in Lemma 28(b).)

Lemma 1 *Let c be a cell returned either by* `announce` *to Line 1 of* `apply`*, or by* `combine` *to Line 7 of* `append`*.*
(a) The value of $c \rightarrow LC$ *is unchanged.*
(b) The value of $c \rightarrow RC$ *is unchanged.*
(c) The value of $c \rightarrow Lop$ *is unchanged.*
(d) The value of $c \rightarrow Op$ *is unchanged.*
(e) Let $c \rightarrow Next = d$, $d \neq \perp$, *at time* t. *Then at any time* t' *such that* $t' \geq t$, $c \rightarrow Next = d$.
(f) Let $c \rightarrow Parent = d$, $d \neq \perp$, *at time* t. *Then at any time* t' *such that* $t' \geq t$, $c \rightarrow Parent = d$.
(g) Let $c \rightarrow Ready = true$, *at time* t. *Then at any time* t' *such that* $t' \geq t$, $c \rightarrow Ready = true$.

Proof
(a),(b) $c \rightarrow LC$ and $c \rightarrow RC$ are assigned values only once: in Line 1 of `announce` or Line 1 of `combine`. This observation implies Lemmas 1(a) and 1(b).
(c),(d) $c \rightarrow Lop$ and $c \rightarrow Op$ are assigned values only once: in Line 1 of `announce` or Line 1 of `combine`. This observation implies Lemmas 1(c) and 1(d).
(e)We first note that $c \rightarrow Next$ is assigned the value $\perp$, only during the initialization of c in `announce` and `combine`, and nowhere else. There are only two places where $c \rightarrow Next$ can take on a non-$\perp$ value: Line 4 of `announce`, and Line 8 of `append`. If process P executes a successful SC($c \rightarrow Next, d$), $d \neq \perp$, in Line 4 of `announce`, then P must have previously executed LL($c \rightarrow Next$) in Line 3, with $\perp$ as the LL operation's return value. This implies Lemma 1(e).

If process P executes a successful SC($c \rightarrow Next, d$), $d \neq \perp$, in Line 8 of `append`, then P must have previously executed LL($c \rightarrow Next$) in Line 2, with $\perp$ as the LL operation's return value (Line 3). This implies Lemma 1(e).

(f)We first note that $c \to Parent$ is assigned the value $\perp$, only during the initialization of c in `announce` and `combine`, and nowhere else. There are only two places where $c \to Parent$ can take on a non-$\perp$ value: Lines 14, 21 of `append`. If process P executes a successful SC($c \to Parent, d$), $d \neq \perp$, in Line 14 (resp. 21) of `append`, then P must have previously executed LL($c \to Parent$) in Line 13 (resp. 20), with $\perp$ as the LL operation's return value. This implies Lemma 1(f).
(g)This follows from the fact that $c \to Ready$ is assigned *false* only during the initialization of c in `combine`, and nowhere else.

□

Lemma 2 *Let c be any cell. There is at most one cell b such that $b \to Next = c$.*

Proof There are only two places where $b \to Next$ (for some b) can be assigned the value c: Line 4 of `announce`, Line 8 of `append`. In both of these places, process P initializes a new cell c and executes SC($b \to Next, c$). Given any cell c, at most one process executes the operation SC($b \to Next, c$) (for some b); further, this process executes the operation once only. Hence, there is at most one cell b such that $b \to Next = c$.

□

At any time t, if a cell c can be reached by the chain of $Next$ pointers, starting at $anchor_{lst}$, then we say that c is *reachable* at position lst at time t. By definition, $anchor_{lst}$ is *not* reachable. Formally,

Definition 1 *Let c be a cell, $1 \leq lst \leq 2n-1$. c is* reachable *at position lst at time t if and only if, at time t:*

- *$anchor_{lst} \to Next = c$, or*
- *$\exists$ a reachable cell b at position lst such that $b \to Next = c$.*

By Lemma 1(c), if c is reachable at time t, then c is reachable at all times after t. We now consider all the cells that are eventually reachable at position lst in a run, and use $List[lst]$ to denote the sequence of such eventually reachable cells. Formally,

Notation 1 *Let $lst, 1 \leq lst \leq 2n-1$, be a position. Let $List[lst]$ denote the list $[List[lst](0), List[lst](1), List[lst](2), \cdots]$, where $\forall k, k \geq 0$, $List[lst](k)$ is defined as follows:*

- *$List[lst](0) = anchor_{lst}$.*
- *$\forall k, k \geq 1$, if $List[lst](k-1)$ is defined, and at some time t, $List[lst](k-1) \to Next = c, c \neq \perp$, then define $List[lst](k) = c$. Otherwise, $List[lst](k)$ is undefined.*

The next Lemma asserts that the cells in the $2n-1$ lists, $List[lst], 1 \leq lst \leq 2n-1$, are all distinct.

Lemma 3 *Let lst, lst' be such that $1 \leq lst, lst' \leq 2n-1$. Let i, j be such that $i \geq 0, j \geq 0$. Then, $(List[lst](i) = List[lst'](j)) \Rightarrow ((lst = lst') \wedge (i = j))$.*

Proof This Lemma follows from Lemma 2.

□

Notation 2 *If c is reachable at lst, then the* position *of c, denoted by $pos(c)$, is defined to be lst.*

By Lemma 3, $pos(*)$ is well-defined.

Definition 2 *Let c be reachable.*

- *If $pos(c) = 1$, c is* root reachable.
- *If $pos(c) > 1$, c is* non-root reachable.
- *If $pos(c) \geq n$, c is* leaf reachable.
- *If $pos(c) < n$, c is* non-leaf reachable.

3.5.1.2 Head

Definition 3 *We say that c* is head *at position lst at time t if and only if $Head[lst] = c$ at t.*

We say that c has been head *at position lst at time t if and only if $\exists t', t' \leq t$, such that c is head at lst at time t'.*

We note that "c has been head at t" says nothing about whether c is head at t. We will now show, in Lemma 5(a), that the initial value of $Head[lst]$ is $List[lst](0)$. The sequence of values that have been successfully written to $Head[lst]$ at time t is:
$List[lst](1), List[lst](2), \cdots, List[lst](k)$, for some $k \geq 0$. As a preliminary step, the next Lemma relates any two successive values of $Head[lst]$.

Lemma 4 *Let $k \geq 0$. Suppose at some time t $Head[lst] = List[lst'](k)$.*[4] *Let c be the first value that is successfully written to $Head[lst]$ after t. Let t' be the time c is successfully written to $Head[lst]$. Then,*

- *$c = List[lst'](k+1)$.*
- *At time τ such that $\tau \geq t'$, $List[lst'](k) \rightarrow Next = List[lst'](k+1)$.*

Proof $Head[lst]$ can be written to in three places: Line 7 of `promote`, Lines 17,24 of `append`. In all these places, $Head[lst]$ changes value from b, the value of $Head[lst]$ returned by the immediately preceding LL($Head[lst]$) (Line 2 of `promote`, Lines 15,22 of `append`), to $c = b \rightarrow Next$ through a successful SC($Head[lst], c$). Further, by Line 4 of `promote`, Lines 12,16 of `append`, Lines 19,23 of `append`, we have $b \rightarrow Next = c \neq \perp$. By Lemma 1(c), once $b \rightarrow Next = c$ holds, $b \rightarrow Next = c$ at all subsequent times. This implies our Lemma.

□

[4] Lemma 5(a) shows that $lst = lst'$.

In Lemma 5(b),(d),(f), we encounter the first instances of invariants in any run, expressed as implications ($\Rightarrow$). Consider Lemma 5(b), which has the form:
(c has been head)$\wedge$(c is not head)$\Rightarrow$ (d has been head).

The interpretation of this implication is: In any run, at any time t, if (c has been head at t) and (c is not head at t), then (d has been head at t). Thus, both sides of the implication ($\Rightarrow$) have an implicit qualification concerning time: They both concern the state of the cells, or other data objects, at a common instant in time.

Lemma 5(d) says that if a cell c is head, then c is either $anchor_{lst}$ or reachable. Lemma 5(e) says that $c \rightarrow Next$ is reachable.

Lemma 5
(a) The initial value of $Head[lst]$ *is* $List[lst](0)$*. The sequence of values that have been successfully written to* $Head[lst]$ *at time t is:*
$List[lst](1), List[lst](2), \cdots, List[lst](k)$*, for some* $k \geq 0$*.*
(b) ($List[lst](i)$ *has been head)*$\wedge$ *(*$List[lst](i)$ *is not head)* $\Rightarrow$ *(*$List[lst](i+1)$ *has been head).*
(c) Let $k \geq 1$*. At the time* $List[lst](k)$ *is successfully written to* $Head[lst]$*,* $List[lst](k)$ *is reachable.*
(d) (c is head at lst)$\Rightarrow$ $(c = anchor_{lst}) \vee$ *(c is reachable at lst).*
(e) Let $c = Head[lst] \rightarrow Next$*, then c is reachable at lst.*
(f) (c is head at lst) $\wedge (c \rightarrow Next = \bot) \wedge$ *(d is reachable at lst)* $\Rightarrow$ *(d has been head at lst)*

Proof
(a) We note that at initialization, $Head[lst] = anchor_{lst} = List[lst](0)$. This statement is a corollary of Lemma 4.
(b) This follows immediately from Part(a).
(c) The proof proceeds by induction on k, and uses Lemma 4. The Induction Step proceeds as follows: Suppose that $List[lst](k-1)$ is reachable (Induction Hypothesis). Then, at the time $List[lst](k)$ is successfully written to $Head[lst]$, $List[lst](k-1) \rightarrow Next = List[lst](k)$ (by Lemma 4). Thus, by Definition 1, $List[lst](k)$ is reachable.
(d) This is a re-statement of Part(c).
(e) We note that $Head[lst]$ is head at lst. By Part(d), we have $(Head[lst] = anchor_{lst}) \vee (Head[lst]$ is reachable at $lst)$. In either case, $c = Head[lst] \rightarrow Next$ is reachable at lst.
(f) By Part(a), there exists a k such that $c = List[lst](k)$. Since $(c \rightarrow Next = \bot) \wedge (d$ is reachable at $lst)$, we have $d = List[lst](m)$, for some m such that $k \geq m \geq 1$. By Part(a), d has been head at lst.

□

Lemma 6 *Let* $b \rightarrow Next = c$*. (c is reachable at lst)* $\Rightarrow$ *(b has been head at lst).*

Proof From the algorithm, $b \rightarrow Next$ is assigned the value c in either Line 4 of `announce`, or Line 8 of `append`. In both these places, a process first executes $head{:=}\ Head[lst']$ for some lst' (Line 2 of `announce`, Line 1 of `append`), and then changes the value of $head \rightarrow Next$ from $\bot$ to c through a successful SC($head \rightarrow Next, c$). After the successful SC($head \rightarrow Next, c$), $Head[lst'] \rightarrow Next = c$. By Lemma 5(e), c is reachable at lst'. Hence, $lst' = lst$. Since

$Head[lst] \rightarrow Next = c$ and $b \rightarrow Next = c$, $Head[lst] = b$ (Lemma 2). In other words, b has been head at lst. This completes our proof.

□

Lemma 6 implies that, if b is head and $b \rightarrow Next = c$ at time t, then $c \rightarrow Next = \perp$ at t. In other words, there is at most one reachable cell beyond the head.

3.5.1.3 Ancestor and Descendant

Definition 4 *Let c be reachable. We say that c is* ready *if and only if $c \rightarrow Ready = true$.*

Notation 3 *Let c be non-leaf reachable. $c = parent(d)$ denotes $(d \neq \perp) \wedge (c \rightarrow LC = d \vee c \rightarrow RC = d)$*

We note that $c = parent(d)$ says nothing about the value of $d \rightarrow Parent$. It is possible that, at time t, $c = parent(d)$ and $d \rightarrow Parent = \perp$.

Definition 5 *Let c be reachable.*

- *We say that c is an* ancestor *of d if and only if:*
  - *$c = d$, or*
  - *$c = parent(d)$, or*
  - *$\exists a$ such that a is an ancestor of d, and $c = parent(a)$.*
- *We say that c is a* root ancestor *of d if and only if:*
  - *c is an ancestor of d, and*
  - *$pos(c) = 1$.*
- *We say that d is a* descendant *of c if and only if c is an ancestor of d.*
- *We say that d is a* leaf descendant *of c if and only if:*
  - *d is a descendant of c, and*
  - *$pos(d) \geq n$.*

We note that, by definition, c is an ancestor, as well as a descendant, of c.

3.5.1.4 Precedence Relations

In this Section, we prove certain precedence relations among the key events of a cell and its parent. These relations form the basis of all subsequent proofs. We provide here an informal and incomplete statement of the result that we aim to prove:

> Let c, d be eventually reachable. Let $c = parent(d)$. Then, the following list specifies the order in which events happen:

1. d becomes reachable.
2. d becomes ready.
3. c becomes reachable.
4. d becomes head.
5. c becomes ready.

(If d is leaf reachable, then 1. and 2. above are concurrent, *i.e.* d becomes ready at the same time it becomes reachable.)

Lemmas 8 to 11 are the rigorous formulations of this informal statement.

Lemma 7 says that if $c = parent(d)$ at time t, then d is reachable and ready at t. Further, $pos(c)$ and $pos(d)$ occupy the appropriate parent and child positions in the binary tree.

Lemma 7 *Let c be non-leaf reachable.*
(a) $c \rightarrow LC = d \neq \perp \Rightarrow$ (d is reachable) $\wedge(pos(d) = 2 * pos(c))\wedge$ (d is ready).
(b) $c \rightarrow RC = d \neq \perp \Rightarrow$ (d is reachable) $\wedge(pos(d) = 2 * pos(c) + 1)\wedge$ (d is ready).

Proof (a) $c \rightarrow LC$ is assigned the value d in `combine` $(d, *, *, *)$, called in Line 7 of `append` (lst). Further, $(d, *)$ is the value returned by `readyOrphan` $(2 * lst)$ (Line 4 of `append` (lst)). In `readyOrphan` $(2 * lst)$, $head = Head[2 * lst]$ (Line 1). $d = head \rightarrow Next$ (Line 2), $d \rightarrow Ready = true$ (Line 4). Therefore, d is reachable at $2 * lst$ (by Lemma 5(e)). Since c is reachable, some process P executed a successful SC($head \rightarrow Next, c$) in Line 8 of `append`. When P executed Line 1 of `append`, $head = Head[lst]$. Therefore, $c = head \rightarrow Next$ is reachable at lst (by Lemma 5(e)). Thus, $pos(c) = lst$, and $pos(d) = 2 * lst = 2 * pos(c)$. Finally, d is ready, since $d \rightarrow Ready = true$.
(b) Similar to the proof of part (a).

□

Lemma 8 says that event 1 (d becomes reachable) and event 2 (d becomes ready) precede event 3 (c becomes reachable).

Lemma 8 *Let c be non-leaf reachable. $c = parent(d) \Rightarrow$ (d is reachable) $\wedge$(d is ready).*

Proof This follows immediately from Lemma 7.

□

Lemma 9 says that event 3 (c becomes reachable) precedes event 4 (d becomes head).

Lemma 9 *Let d be non-root reachable. (d has been head)$\Rightarrow \exists$ reachable c such that $c = parent(d)$.*

Proof d becomes head in either Line 17 or Line 24 of `append`. Consider the case of Line 17. (The case of Line 24 is analogous.) Let P be the process that executed a successful SC($Head[2*lst], d$), where $d = lchild$, in Line 17, thus causing d to become head. We note that during P's execution of `append`, $lchild = newcell \rightarrow LC$ (Line 11), $newcell = head \rightarrow Next$ (Line 2 or 9), $head = Head[lst]$ (Line 1). Let c in our Lemma be $newcell$. Then, by Lemma 5(e), c is reachable. By Notation 3, $c = parent(d)$ since $c \rightarrow LC = lchild = d$. This completes the proof.

□

Lemma 10 says that event 2 (d becomes ready) precedes event 4 (d becomes head). Lemma 10 applies not only to non-root reachable d (as indicated by the informal statement given at the beginning of this section), but also to root reachable d.

Lemma 10 *Let d be such that $\forall lst, 1 \leq lst \leq 2n-1, d \neq anchor_{lst}$. ($d$ has been head) $\Rightarrow$ (d is ready).*

Proof Since $\forall lst, 1 \leq lst \leq 2n-1, d \neq anchor_{lst}$, and d has been head, d is reachable (by Lemma 5(d)). If d is non-root reachable, Lemma 10 is an immediate corollary of Lemmas 8 and 9.

Consider the case in which d is root reachable. d becomes $Head[1]$ through a successful SC($Head[1], d$) in Line 7 of `promote` (1). Further, $d \rightarrow Ready = true$ (Line 5). Hence, d is ready.

□

Lemma 11 says that event 4 (d becomes head) precedes event 5 (c becomes ready).

Lemma 11 *Let c be non-leaf reachable. Then, (c is ready)$\wedge$($c = parent(d)$) $\Rightarrow$ (d has been head).*

Proof Consider the case of $c \rightarrow LC = d$. (The case of $c \rightarrow RC = d$ is analogous.) c becomes ready through process P executing Line 25 of `append`. Further, since $lchild = c \rightarrow LC = d \neq \perp$ (Lines 11,12), P executes Line 15 before executing Line 25. At the time t when P executes Line 15, $c = parent(d)$ holds true. By Lemma 8, ($c = parent(d)$) $\Rightarrow$ (d is reachable). Let x be such that $x \rightarrow Next = d$. By Lemma 6, x has been head at $pos(d)$ at time t.

If x is not head at t, then, by Lemma 5(b), d has been head at t. Therefore our Lemma holds in this case.

Suppose x is head at t. Since $lchild = d$ (Line 11), P observes that $lchead \rightarrow Next = lchild$ in Line 16, and executes SC($Head[pos(d)], d$) in Line 17. If the SC operation returns *true*, then d has been head when P completes Line 17, hence also when P completes Line 25. If the SC operation returns *false*, then by Lemma 5(a), d has been head when P begins executing Line 17. In either case, d has been head when P executes Line 25. This completes the proof.

□

By Lemma 10, event 2 (c becomes ready) precedes event 4 (c becomes head). By Lemma 11, event 4 (d becomes head) precedes event 5 (c becomes ready). Combining these two results, we have the next Lemma: (d becomes head) precedes (c becomes head).

Lemma 12 *Let c be non-leaf reachable. (c has been head)$\wedge$($c = parent(d)$) $\Rightarrow$ (d has been head).*

Proof By Lemma 10, (c has been head) $\Rightarrow$ (c is ready). By Lemma 11, (c is ready)$\wedge$($c = parent(d)$) $\Rightarrow$ (d has been head). Thus, our Lemma holds.

□

3.5.1.5 Uniqueness of Parent

The next Lemma asserts the uniqueness of reachable cell b such that $b = parent(d)$.

Lemma 13 *Let b, c be non-leaf reachable.* ($b = parent(d)$) $\wedge$ ($c = parent(d)$) $\Rightarrow$ ($b = c$).

Proof For contradiction, assume $b \neq c$. By Lemma 7, $pos(b) = pos(c)$. Let $b = List[pos(b)](k), c = List[pos(b)](l)$. Without loss of generality, let $k < l$. Let P be the process that executes a successful SC($head \rightarrow Next, c$) in Line 8 of `append`. Let t be the time P executes Line 1 of this `append`. Thus, at t, $Head[pos(b)] = List[pos(b)](l-1)$. Since $k < l$, b has been head at t. By Lemma 12, since $b = parent(d)$, d has been head at t.

$c = parent(d)$ implies that when P executes `readyOrphan` ($pos(d)$) in Line 4 or 5 of `append` at some time after t , $(d, *)$ is returned. This in turn implies that when P executes `readyOrphan` ($pos(d)$), $head = Head[pos(d)]$ in Line 1, and $head \rightarrow Next = d$ in Line 2. In other words, d has not been head when P executes Line 1 of `readyOrphan` ($pos(d)$). Thus, d has not been head at t. This contradiction proves that $b = c$.

□

Lemma 14 asserts that, if d has been head, then:

- there is a unique c such that $c = parent(d)$ (uniqueness is proved in Lemma 13),
- c is reachable (this corresponds to event 3 (c becomes reachable) preceding event 4 (d becomes head) in the informal statement at the beginning of the previous section),
- $d \rightarrow Parent$ holds the proper value c.

Lemma 14 *Let d be non-root reachable. (d has been head)$\Rightarrow \exists$ unique non-leaf reachable c such that $(c = parent(d)) \wedge (d \rightarrow Parent = c)$.*

Proof By Lemmas 9, (d has been head)$\Rightarrow \exists$ reachable c such that $c = parent(d)$. We now show that c is non-leaf. By Lemma 7, since d is non-root reachable, and $pos(d) = 2 * pos(c)$, we conclude that c is non-leaf reachable. By Lemma 13, such a c is unique. It remains to prove that (d has been head)$\Rightarrow$ ($d \rightarrow Parent = c$).

We consider the case of $c \rightarrow LC = d$. (The case of $c \rightarrow RC = d$ is analogous.) Suppose that d becomes head by process P executing a successful SC($Head[pos(d)], d$) in Line 17 of `append`. This implies that $lchild = d$ (Line 17). In Line 11, $lchild = newcell \rightarrow LC$. By Lemma 13, $newcell = c$. We note that P executes Line 13 before executing Line 17 (or 24, in the case of $c \rightarrow RC = d$). If $lchild \rightarrow Parent = d \rightarrow Parent = \perp$ (Line 13), then P executes SC($d \rightarrow Parent, c$) (Line 14). If the SC operation returns *true*, then $d \rightarrow Parent = c$ after P executes Line 14. Our Lemma is thus proved.

However, if either $d \rightarrow Parent \neq \perp$ (Line 13) or the SC operation in Line 14 returns *false*, then $d \rightarrow Parent \neq \perp$ before P executes Line 15. $d \rightarrow Parent$ is assigned a non-$\perp$ value only through some process P' executing a successful SC($d \rightarrow Parent, newcell$) in Line 14 of `append`. However, when P' executes Line 11, $d = newcell \rightarrow LC$. By Lemma 13, $newcell = c$. Therefore, the non-$\perp$ value assigned by P' to $d \rightarrow Parent$ is c. This completes the proof.

□

3.5.1.6 Properties of `append`

Lemma 15 says the following: Suppose that the head at the child position lst is b, and the head at the parent position $\lfloor lst/2 \rfloor$ is c at time t. Suppose further that at some subsequent time t', b is no longer head. Then, $List[\lfloor lst/2 \rfloor]$ must have grown beyond c at t'.

Lemma 15 *Let* $lst > 1$*. Suppose at time* t*,* $Head[lst] = b$*,* $Head[\lfloor lst/2 \rfloor] = c$*. Then, (*$b$ *has been head)*$\wedge$*(*b *is not head)*$\Rightarrow$ $(c \rightarrow Next \neq \perp)$*.*

Proof For contradiction, suppose that at time t', (b has been head)$\wedge$ (b is not head)$\wedge$($c \rightarrow Next = \perp$). By Lemma 5(a), $t' > t$. By Lemma 5(b), there exists a reachable d such that $b \rightarrow Next = d$, and d has been head at t'. By Lemma 9, there exists a reachable a such that $a = parent(d)$ at t'. Since c is head, $c \rightarrow Next = \perp$, and a is reachable at $pos(c) = \lfloor lst/2 \rfloor$ at t', we conclude that a has been head at t' (Lemma 5(f)). Since c is head at both t and t', a has been head at t. By Lemma 12, (a has been head)$\wedge$($a = parent(d)$) $\Rightarrow$ (d has been head) at t. This contradicts the assumption that b is head at t. Our Lemma is thus proved by this contradiction.

□

Lemma 16 is the key Lemma regarding the property of `append`. It says the following: Let t be *any* time before the invocation of `append`($\lfloor lst/2 \rfloor$). Suppose that at time t, the head at the child position lst is b, the head at the parent position $\lfloor lst/2 \rfloor$ is c, and $b \rightarrow Next$ is ready. Then, after `append`($\lfloor lst/2 \rfloor$) terminates, $c \rightarrow Next$ is ready. We note that $c \rightarrow Next$ could have become ready *before* the invocation of `append`($\lfloor lst/2 \rfloor$). Our Lemma allows for such a possibility.

Lemma 16 *Suppose that at time* t *before process* P *invokes* `append`*(*$\lfloor lst/2 \rfloor$*),* $Head[lst] = b, b \rightarrow Next = d \neq \perp, d$ *is ready,* $Head[\lfloor lst/2 \rfloor] = c$*. Then, at any time after* `append`*(*$\lfloor lst/2 \rfloor$*) terminates,* $\exists$ *reachable* e *such that* $(c \rightarrow Next = e) \wedge (e$ *is ready).*

Proof We note that c is head at t. If at any time during the execution of `append`($\lfloor lst/2 \rfloor$) c is not head, then by Lemma 5(b), $\exists$ reachable e such that ($c \rightarrow Next = e$), and e has been head. By Lemma 10, e is ready. Our Lemma is then proved.

We now consider the case where c is head throughout the execution of `append`($\lfloor lst/2 \rfloor$). Consider process P executing `append`($\lfloor lst/2 \rfloor$). In Line 1, $head = Head[\lfloor lst/2 \rfloor] = c$.

Case 1 Suppose $\exists$ reachable e such that $c \rightarrow Next = e \neq \perp$ in Line 2.

We note that in this Case, Line 25 is always executed. Further, $newcell = e$ (Line 2). This implies that, at the time P completes Lines 25,, $e \rightarrow Ready = true$. By Lemma 1(e), our Lemma holds.

Case 2 Suppose $c \rightarrow Next = \perp$ in Line 2.

Consider P executing `readyOrphan` (lst) (Line 4 or 5 of `append`). Let t' be the time P executes Line 1 of `readyOrphan` (lst).

Case 2a Suppose $Head[lst] \neq b$ at t'.

By Lemma 15, since (b has been head)$\wedge$ (b is not head) at t', we have $c \rightarrow Next \neq \perp$ at t'. Therefore, there exists a reachable e such that $newcell = c \rightarrow Next = e \neq \perp$ in Line 9 of `append`. This implies that P always executes Line 25. By the argument used in Case 1, at the time P completes Lines 25, $e \rightarrow Ready = true$. Hence, our Lemma holds.

Case 2b Suppose $Head[lst] = b$ at t'.

By the premise of our Lemma, in Lines 1-4 of `readyOrphan` (lst), $newcell = b \rightarrow Next = d \neq \perp, newcell \rightarrow Ready = true$. Further, `readyOrphan` ($lst$) returns $(b, *)$ to Lines 4 or 5 of `append`. P then executes SC($c \rightarrow Next, *$) in Line 8 of `append`. Whether the SC operation returns *true* or *false*, $newcell = c \rightarrow Next \neq \perp$ when P executes Line 9. By the argument used in Case 2a, our Lemma holds in this case, too.

□

3.5.1.7 Properties of `promote`

In this section, we prove the key property of `promote`. Lemma 17 concerns the property of `promote`(1). It asserts that if a cell is ready before `promote`(1), then it has been head after `promote`(1).

Lemma 17 *Let r be root reachable and ready before the invocation of* `promote`*(1). Then, after* `promote`*(1) terminates, r has been head.*

Proof Let s be root reachable or $anchor_1$, such that $s \rightarrow Next = r$. By Lemma 6, since r is reachable, s has been head before process P invokes `promote`(1). Suppose that at some time t during P's execution of `promote`(1), $Head[1] \neq s$. By Lemma 5(b), r has been head at t. Thus, our Lemma holds.

We now consider the case where $Head[1] = s$ throughout the execution of `promote`(1). This implies that $head = Head[1] = s$ (Line 2 of `promote`(1)). Since $newcell = s \rightarrow Next = r \neq \perp$ (Line 3), $newcell \rightarrow Ready = r \rightarrow Ready = true$ (Line 5), P executes SC($Head[1], r$) in Line 7. Whether the SC operation returns *true* or *false*, $Head[1] \neq s$ after P's execution of Line 7 (Lemma 4). This contradicts the assumption that $Head[1] = s$ throughout the execution of `promote`(1). The proof is now complete.

□

Lemma 18 asserts that if a cell d is ready before `promote` ($pos(d)$) (for any $pos(d)$), then d has been head after `promote` ($pos(d)$) completes. Further, if d is non-root reachable, then d has a ready parent after `promote` ($pos(d)$) completes.

Lemma 18 *Let d be reachable and ready before the invocation of* `promote` *($pos(d)$). Then, after* `promote` *($pos(d)$) terminates,*
(a) d has been head,
(b) if d is non-root reachable, $\exists$ non-leaf reachable c such that $c = parent(d) \wedge (c$ is ready$)$.

Proof The proof is by induction on $i, 0 \leq i \leq \log n$, such that $2^i \leq pos(d) \leq 2^{i+1} - 1$. (If $i = 0$, d is root reachable. If $i = \log n$, d is leaf reachable.)
Induction Basis $i = 0$.

This Lemma for $i = 0$ is identical to Lemma 17.
Induction Step Suppose Lemma 18 holds for all d such that $1 \leq pos(d) \leq 2^i - 1$, where $1 \leq i \leq \log n$. We now show that Lemma 18 holds for d such that $2^i \leq pos(d) \leq 2^{i+1} - 1$.

Let a be such that $a \rightarrow Next = d$. Let t_0 be the earliest time when d is reachable and ready. By Lemma 6, since d is reachable, a has been head at t_0. We now show that a is head at t_0. Suppose a is not head at t_0. By Lemma 5(b), d has been head at t_0. By Lemma 10, d becomes ready before d becomes head. Therefore, d becomes head after t_0. This contradiction proves that a is head at t_0.

We note that d is non-root reachable. Let $Head[\lfloor pos(d)/2 \rfloor] = e$ at t_0. As shown above, $Head[pos(d)] = a$, $a \rightarrow Next = d, d \rightarrow Ready = true$ at t_0. By assumption, t_0 is a time before process P invokes `promote`($pos(d)$). In `promote`($pos(d)$), P invokes `append`($\lfloor pos(d)/2 \rfloor$) in Line 9 after t_0. By Lemma 16, after P completes `append`($\lfloor pos(d)/2 \rfloor$) in Line 9, there is a reachable f such that ($e \rightarrow Next = f$) and (f is ready). P next invokes `promote`($\lfloor pos(d)/2 \rfloor$) in Line 10.

Applying the Induction Hypothesis, we observe that: Since f is reachable and ready before P invokes `promote`$(\lfloor pos(d)/2 \rfloor)$ in Line 10, f has been head when P completes `promote`$(\lfloor pos(d)/2 \rfloor)$ in Line 10. Let t_1 be the earliest time when $Head[\lfloor pos(d)/2 \rfloor] = f$.

Case 1 a is not head at t_1.

We note that a is head at t_0, and is not head at t_1. By Lemma 5(b), d has been head at t_1. Thus, part (a) of our Lemma holds. By Lemma 9, there exists a reachable x at $\lfloor pos(d)/2 \rfloor$ such that $x = parent(d)$ at t_1. Since t_1 is the earliest time when $Head[\lfloor pos(d)/2 \rfloor] = f$, $f \rightarrow Next = \perp$ at t_1. By Lemma 5(f), x has been head at t_1. By Lemma 10, x is ready at t_1. Thus, x is non-leaf reachable, $x = parent(d)$ and x is ready at t_1. Hence, part (b) of our Lemma holds.

Case 2 a is head at t_1.

In this case, $Head[\lfloor pos(d)/2 \rfloor] = f, Head[pos(d)] = a, a \rightarrow Next = d, d \rightarrow Ready = true$ at t_1. t_1 is a time before P invokes `append`$(\lfloor pos(d)/2 \rfloor)$ in Line 11. By Lemma 16, when P completes `append`$(\lfloor pos(d)/2 \rfloor)$ in Line 11, there exists a reachable g such that ($f \rightarrow Next = g$) and (g is ready).

Let P' be the process that executes the successful SC$(f \rightarrow Next, g)$ in Line 8 of `append`. Prior to Line 8, P' executes `readyOrphan`$(pos(d))$ in Lines 4,5 of `append`. Let τ be any time when P' is executing `readyOrphan`$(pos(d))$. We note that $f \rightarrow Next = \perp$ at τ. Suppose $Head[pos(d)] \neq a$ at τ. Let t in Lemma 15 be t_1. Since (a has been head)$\wedge$ (a is not head) at τ, we conclude that $f \rightarrow Next \neq \perp$ at τ (Lemma 15). This contradiction proves that at any time τ when P' is executing `readyOrphan`$(pos(d))$, $Head[pos(d)] = a$. Hence, as P' executes Line 1 of `readyOrphan`$(pos(d))$, $head = a$. Finally, `readyOrphan`$(pos(d))$ returns $(d, d \rightarrow Op)$ to Line 4, or 5 of `append`$(\lfloor pos(d)/2 \rfloor)$. This in turn implies that in `combine` (called in Line 7 of `append`), $(g \rightarrow LC = d) \vee (g \rightarrow RC = d)$. In other words, $g = parent(d)$. As was proved above, when P completes Line 11 of `promote`$(pos(d))$, g is ready. So, g is non-leaf reachable, $g = parent(d)$, and g is ready. Part(b) of our Lemma is thus proved.

By Lemma 11, d has been head when P completes `promote`$(pos(d))$. Thus, part (a) of our Lemma holds.

□

3.5.1.8 Properties of `apply`

We are now in a position to prove (in Lemma 21) the crucial property of `apply` that we need. In Lemma 19, we prove the property of `apply`, assuming that *opcell* is reachable and ready, after Line 1 of `apply` completes. In Lemma 20, we prove that this assumption indeed holds. Thus, in Lemma 21, the property of `apply` holds without any assumption.

Lemma 19 asserts the following: Assume that *opcell* is reachable and ready after Line 1 of `apply` completes. Then, after the *for* loop in Lines 2-3 completes, at each of the $\log n + 1$ positions along the path from *opcell* to the root, there is a unique ancestor of *opcell*. Thus, *opcell* has $\log n + 1$ ancestors (including *opcell* itself, and a cell at the root position). Further, each ancestor of *opcell* (including *opcell* itself) has been head.

Lemma 19 *Consider process P executing* `apply`*$(P, op, \mathcal{O})$. Suppose that when P completes Line 1, opcell is reachable and ready. Then, after P exits the for loop in Lines 2-3, $\forall i, 0 \leq i \leq \log n$, $\exists$ a reachable a_i, such that:*

- *a_i is a unique ancestor of opcell at position $\lfloor (n+P)/2^i \rfloor$,*
- *a_i has been head,*
- *$\forall i < \log n, a_i \rightarrow Parent = a_{i+1}$.*
- *$a_0 = opcell$.*

Proof By Definition 5 *opcell* is an ancestor of *opcell* at position $n+P$. Let $a_0 = opcell$. If a_i is an ancestor of *opcell*, then $a_{i+1} = parent(a_i)$ is (by Definition 5) an ancestor of *opcell* too. By Lemma 18, we have:

> (*) $\forall 0 \leq i \leq \log n$: Let a_i be reachable at $\lfloor (n+P)/2^i \rfloor$ and ready before the invocation of `promote` ($\lfloor (n+P)/2^i \rfloor$). Then, after `promote` ($\lfloor (n+P)/2^i \rfloor$) terminates,
>
> - a_i has been head,
> - $\forall i < \log n, \exists a_{i+1}$ such that a_{i+1} is reachable at $\lfloor (n+P)/2^{i+1} \rfloor$, ready, and $a_{i+1} = parent(a_i)$.

We note that a_0 is reachable at $n+P$, and ready before the *for* loop in Lines 2,3 of `apply`$(P, op, \mathcal{O})$ begins. In the *for* loop, P executes `promote` ($\lfloor (n+P)/2^i \rfloor$), $\forall i$ such that $0 \leq i \leq \log n$.

By $\log n + 1$ applications of (*) above, we get: $\forall i, 0 \leq i \leq \log n : \exists$ reachable a_i such that a_i is an ancestor of *opcell* at $\lfloor (n+P)/2^i \rfloor$, and a_i has been head.

By Lemma 13, there exists at most one ancestor of *opcell* at $\lfloor (n+P)/2^i \rfloor$, for any given i. This proves the uniqueness of a_i as stated in our Lemma.

Let $i < \log n$. By Lemma 14, a_i has been head implies that $a_i \rightarrow Parent = a_{i+1}$.

□

Lemma 20 proves that the assumption in Lemma 19 indeed holds. The inductive proof of Lemma 20 uses Lemma 19.

Lemma 20 *Consider process P executing* `apply`*$(P, op, \mathcal{O})$. When P completes Line 1, opcell is reachable and ready, and $opcell \rightarrow Op = op$.*

Proof The proof is by induction on the successive invocations of `apply`(P, *, O) by P in a run.
Induction Basis Consider the first invocation of `apply`(P, *, O) by P. Let P invoke `apply`$(P, op, \mathcal{O})$. In Line 2 of `announce`(op, P) (called in Line 1 of `apply`$(P, op, \mathcal{O})$), $head = Head[n+P] = anchor_{n+P}$. We note that during initialization, $anchor_{n+P} \rightarrow Next := \perp$, and that the only way $anchor_{n+P} \rightarrow Next$ can be assigned a new value is when P executes SC($head \rightarrow Next, *$) in Line 4 of `announce`. Therefore the SC operation in Line 4 returns *true*, and *opcell* is reachable when P completes `announce`(op, P). By Line 1 of `announce`(op, P), we have $opcell \rightarrow Ready = true, opcell \rightarrow Op = op$. Thus, our Lemma holds.
Induction Step The induction hypothesis is that Lemma 20 holds for all previous invocations of `apply`(P,*, O) by P. We now prove that Lemma 20 holds with respect to the current invocation.

Let the invocation of `apply`(P, *, O) by P that immediately precedes the current invocation, `apply`$(P, op, \mathcal{O})$, be `apply`$(P, op', \mathcal{O})$. Let $c' = opcell$, the value returned in Line 1 of

$\texttt{apply}(P, op', \mathcal{O})$. By Lemma 19, c' has been head when P completes $\texttt{apply}(P, op', \mathcal{O})$. Let t be the time when P executes Line 3 of $\texttt{announce}(op, P)$ (called in Line 1 of $\texttt{apply}(P, op, \mathcal{O})$). Since the only way $c' \rightarrow Next$ can be assigned a non-$\perp$ value is by process P executing a successful SC($head \rightarrow Next, *$) in Line 4 of `announce`, $c' \rightarrow Next = \perp$ at t. This in turn implies that c' is head at t, and hence also at the time P executes Line 2 of $\texttt{announce}(op, P)$. Hence P executes SC($head \rightarrow Next, opcell$) in Line 4 of $\texttt{announce}(op, P)$. Further, the SC operation returns *true*. This implies that *opcell* is reachable after P completes Line 1 of $\texttt{apply}(P, op, \mathcal{O})$. By Line 1 of $\texttt{announce}(op, P)$, *opcell* is ready, and $opcell \rightarrow Op = op$. The proof is now complete.

□

Lemma 21 is the key result in this section. It is the statement of Lemma 19, without the assumption that when P completes Line 1, *opcell* is reachable and ready.

Lemma 21 *Consider process P executing* $\texttt{apply}(P, op, \mathcal{O})$*. After P exits the* for *loop in Lines 2-3, $\forall i, 0 \leq i \leq \log n$, $\exists$ a reachable a_i, such that:*

- *a_i is a unique ancestor of opcell at position $\lfloor (n + P)/2^i \rfloor$,*
- *a_i has been head,*
- *$\forall i < \log n, a_i \rightarrow Parent = a_{i+1}$.*
- *$a_0 = opcell$.*

Proof This Lemma is a corollary of Lemmas 19 and 20.

□

3.5.2 Proof of Linearizability

The objective of this section is to prove that our algorithm is linearizable. As a first step, we define formally the notion of *linearizability*.

Definition 6 *Consider a run $\mathcal{R}$.*

- INVOCATIONS*($\mathcal{R}$) denotes the set of invocations of* `apply` *in $\mathcal{R}$.*
- LEAFCELLS*($\mathcal{R}$) denotes the set of leaf reachable cells in $\mathcal{R}$.*
- *An invocation of* $\texttt{apply}(P, op, \mathcal{O})$ *in $\mathcal{R}$ is* complete *if the procedure* $\texttt{apply}(P, op, \mathcal{O})$ *terminates in $\mathcal{R}$.*

Definition 7 *The run $\mathcal{R}$ is* linearizable *if there exists a sequence, invocSeq($\mathcal{R}$), of invocations in* INVOCATIONS($\mathcal{R}$)*, such that:*

- *each invocation in* INVOCATIONS($\mathcal{R}$) *appears at most once in invocSeq($\mathcal{R}$).*
- *each complete invocation in* INVOCATIONS($\mathcal{R}$) *appears exactly once in invocSeq($\mathcal{R}$).*

- *Let invocations* `apply`$(P, op, \mathcal{O})$*,* `apply`$(P', op', \mathcal{O})$ *be in* $invocSeq(\mathcal{R})$*. If procedure* `apply`$(P, op, \mathcal{O})$ *terminates before procedure* `apply`$(P', op', \mathcal{O})$ *begins, then invocation* `apply`$(P, op, \mathcal{O})$ *precedes invocation* `apply`$(P', op', \mathcal{O})$ *in* $invocSeq(\mathcal{R})$*.*

- *Let* $\mathcal{R}'$ *be a run in which the invocations in* $invocSeq(\mathcal{R})$ *are applied sequentially (without overlap) to* $\mathcal{O}$*. Suppose that procedure* `apply`$(P, op, \mathcal{O})$ *returns* x *in* $\mathcal{R}$*. Then, procedure* `apply`$(P, op, \mathcal{O})$ *returns* x *in* $\mathcal{R}'$*.*

3.5.2.1 Functions *invoca* and *leaf*

Definition 8 *We define a function* invoca*:* LEAFCELLS$(\mathcal{R}) \rightarrow$ INVOCATIONS$(\mathcal{R})$ *thus:*

> invoca(c) = (`apply`$(P, op, \mathcal{O})$) *if and only if* `announce` *called in Line 1 of* `apply`$(P, op, \mathcal{O})$ *produces a leaf reachable cell* c *in* $\mathcal{R}$

Since each leaf reachable cell produced in `announce` is a new cell, *invoca* is well-defined. Further, since each invocation produces at most one leaf reachable cell, *invoca* is one-to-one. Let INVOCATIONSWITHCELLS$(\mathcal{R})$ denote the range of *invoca*.

Definition 9 *We define a function* leaf*:* INVOCATIONSWITHCELLS$(\mathcal{R}) \rightarrow$ LEAFCELLS$(\mathcal{R})$ *thus:*

> leaf(`apply`$(P, op, \mathcal{O})$)=c *if and only if* invoca(c) = (`apply`$(P, op, \mathcal{O})$).

Since *invoca* is one-to-one, *leaf* is well-defined.

We note that *leaf*(`apply`$(P, op, \mathcal{O})$) is the cell returned by `announce`(op, P) called in Line 1 of `apply`$(P, op, \mathcal{O})$.

Lemma 22 *Let* $c =$ leaf(`apply` $(P, op, \mathcal{O})$)*. Then,* $c \rightarrow Op = op$*.*

Proof This is an immediate corollary of Lemma 20.

□

3.5.2.2 Leaf Sequence

Notation 4

- $[x_1, x_2, x_3, \cdots]$ *denotes a sequence whose* i*th element is* x_i*.*

- *Let* σ_1 *be a finite sequence, and* σ_2 *be a sequence. Then,* $\sigma_1 \circ \sigma_2$ *denotes the concatenation of* σ_1 *and* σ_2*.*

- ϵ *denotes the empty sequence.*

We define the *leaf sequence* of a reachable cell c, denoted by *leafSeq*(c), to be the sequence of the leaf descendants of c, in the left-to-right order. Formally,

Definition 10 *Let c be $\perp$ or a reachable cell. We define the* leaf sequence *of c, denoted by leafSeq(c), to be a sequence of leaf reachable cells satisfying the following:*
(a) If $c = \perp$, then leafSeq(c) $= \epsilon$;
(b) If c is leaf reachable, then leafSeq(c) $= [c]$;
(c) If c is non-leaf reachable, then leafSeq(c) $=$ leafSeq($c \rightarrow LC$) $\circ$ leafSeq($c \rightarrow RC$).

By Lemma 7, if c is non-leaf reachable, then $c \rightarrow LC$ (and $c \rightarrow RC$) is either $\perp$ or reachable. Thus, (c) in the above definition is well-defined.

□

Let $\mathcal{R}$ be a run. We define the *leaf sequence* of $\mathcal{R}$, denoted by *leafSeq*($\mathcal{R}$), to be the concatenation of the leaf sequences of the root reachable cells in $\mathcal{R}$.

Definition 11 *Let $\mathcal{R}$ be a run. Recall that $List[1] = [r_0, r_1, r_2, \cdots]$, where $r_0 = anchor_1$, and $\forall i > 0, r_{i-1} \rightarrow Next = r_i$.*

Define the leaf sequence *of $\mathcal{R}$, denoted by leafSeq($\mathcal{R}$), as follows: leafSeq($\mathcal{R}$)=leafSeq(r_1) $\circ$ leafSeq(r_2) $\circ$ leafSeq(r_3) $\circ \cdots$.*

Thus, leafSeq($\mathcal{R}$) is the sequence of leaf cells of the root reachable cells in a run $\mathcal{R}$, in the left-to-right order.

Definition 12 *Let c be a reachable cell in a run $\mathcal{R}$. The* prior leaf sequence *of c, denoted by priorLeafSeq(c), is defined to be the sequence α of leaf reachable cells such that $\alpha \circ$ leafSeq(c) is a prefix of leafSeq($\mathcal{R}$).*

Let c be a reachable cell in a run $\mathcal{R}$. By Definitions 10, 11, and Lemma 21, *leafSeq*(c) occupies a unique position in *leafSeq*($\mathcal{R}$). α is therefore well-defined in the above Definition.

We note that *priorLeafSeq*(c) is the sequence of leaf reachable cells (in the left-to-right order) up to, but excluding, the left-most leaf descendant of c.

3.5.2.3 Invocation Sequence

Definition 13 *Let leafSeq($\mathcal{R}$) $= [c_1, c_2, c_3, \cdots]$. Define the* invocation sequence *of $\mathcal{R}$, invocSeq($\mathcal{R}$) $=$* $[\text{invoca}(c_1), \text{invoca}(c_2), \cdots]$.

The next three Lemmas are important in our proof of linearizability.

Lemma 23 *Each invocation in* INVOCATIONS($\mathcal{R}$) *appears at most once in invocSeq($\mathcal{R}$).*

Proof Each invocation produces at most one leaf reachable cell c. By Lemma 21, c has at most one root ancestor. Thus, c appears at most once in *leafSeq*($\mathcal{R}$). Therefore, each invocation appears at most once in *invocSeq*($\mathcal{R}$).

□

Lemma 24 *Each complete invocation in* INVOCATIONS($\mathcal{R}$) *appears exactly once in invocSeq($\mathcal{R}$).*

Proof Each complete invocation produces exactly one leaf reachable cell c. By Lemma 21, c has exactly one root ancestor. Thus, c appears exactly once in $leafSeq(\mathcal{R})$. Therefore, each complete invocation appears exactly once in $invocSeq(\mathcal{R})$.

□

Lemma 25 *Let invocations* $\texttt{apply}(P, op, \mathcal{O})$, $\texttt{apply}(P', op', \mathcal{O})$ *be in* $invocSeq(\mathcal{R})$. *If procedure* $\texttt{apply}(P, op, \mathcal{O})$ *terminates before procedure* $\texttt{apply}(P', op', \mathcal{O})$ *begins, then invocation* $\texttt{apply}(P, op, \mathcal{O})$ *precedes invocation* $\texttt{apply}(P', op', \mathcal{O})$ *in* $invocSeq(\mathcal{R})$.

Proof Let $List[1] = [r_0, r_1, \cdots]$. By Lemma 21, $leaf(\texttt{apply}(P, op, \mathcal{O}))$ has a root ancestor r_i before $\texttt{apply}(P, op, \mathcal{O})$ terminates. By Lemma 7, and from the algorithm, $leaf(\texttt{apply}(P', op', \mathcal{O}))$ has a root ancestor r_j after $\texttt{apply}(P', op', \mathcal{O})$ begins. Thus, $i < j$. Hence, $leaf(\texttt{apply}(P, op, \mathcal{O}))$ precedes $leaf(\texttt{apply}(P', op', \mathcal{O}))$ in $leafSeq(\mathcal{R})$. As a result, $\texttt{apply}(P, op, \mathcal{O})$ precedes $\texttt{apply}(P', op', \mathcal{O})$ in $invocSeq(\mathcal{R})$.

□

3.5.2.4 Operation Sequence

Definition 14 *Let c be reachable.*

- *Let* $leafSeq(c)$ *be* $[c_1, c_2, c_3, \cdots, c_k]$. *We define the* operation sequence *of c,* $opSeq(c) = [c_1 \rightarrow Op, c_2 \rightarrow Op, c_3 \rightarrow Op, \cdots, c_k \rightarrow Op]$.
- *Let* $priorLeafSeq(c)$ *be* $[c_1, c_2, c_3, \cdots, c_k]$. *We define the* prior operation sequence *of c,* $priorOpSeq(c) = [c_1 \rightarrow Op, c_2 \rightarrow Op, c_3 \rightarrow Op, \cdots, c_k \rightarrow Op]$.

Thus, $opSeq(c)$ is the sequence of operations in the Op fields of the cells of $leafSeq(c)$. $priorOpSeq(c)$ is the sequence of operations in the Op fields of the cells of $priorLeafSeq(c)$.

3.5.2.5 *Op* Field of a Cell

We define $\delta^*_{state}(s, [op_1, op_2, \cdots, op_k])$ to be the state of the object $\mathcal{O}$ after applying successively the operations $op_1, op_2, op_3, \cdots, op_k$ to $\mathcal{O}$ in state s. Formally,

Definition 15 *Let* OP^+ *be the set of sequences of one or more operations. Define* $\delta^*_{state} : Q \times OP^+ \rightarrow Q$ *as follows:*

Let $s \in Q$ *be a state. Let* $op \in OP, \gamma \in OP^+$. *Then,*

- $\delta^*_{state}(s, [op]) = \delta_{state}(s, op)$.
- $\delta^*_{state}(s, \gamma \circ [op]) = \delta_{state}(\delta^*_{state}(s, \gamma), op)$.

We note that given two sequences of operations γ_1, γ_2, $\delta^*_{state}(\delta^*_{state}(s, \gamma_1), \gamma_2) = \delta^*_{state}(s, \gamma_1 \circ \gamma_2)$.

Recall that $op' \bigotimes op'' = op$ if, for any state s, applying operation op to the implemented object $\mathcal{O}$ in state s results in the same state as applying first op', then op'' to $\mathcal{O}$ in state s. We note that $\bigotimes$ is associative. Thus, $((op_1 \bigotimes op_2) \bigotimes op_3) \bigotimes op_4 = op_1 \bigotimes ((op_2 \bigotimes op_3) \bigotimes op_4)$.

We define $\biguplus$ on a non-trivial sequence of operations, such that $\biguplus([op_1, op_2, \cdots, op_k])$ is the operation that results from combining, using$\bigotimes$, the operations $op_1, op_2, \cdots, op_k$. Formally,

Definition 16 *Let* $[op_1, op_2, \cdots, op_k]$ *be a non-trivial sequence of operations. Define* $\biguplus([op_1, op_2, \cdots, op_k])$ *as follows:*

- *If* $k = 1$, $\biguplus([op_1]) = op_1$,
- *If* $k > 1$, $\biguplus([op_1, op_2, \cdots, op_k]) = ((\cdots((op_1 \bigotimes op_2) \bigotimes op_3) \cdots \bigotimes op_k)$.

By the property of $\bigotimes$, we have: Let s be any state. Let σ be a non-trivial sequence of operations. Then, $\delta_{state}(s, \biguplus(\sigma)) = \delta^*_{state}(s, \sigma)$.

Lemma 26 says that $c \rightarrow Op$ is the operation that results from combining, using $\bigotimes$, the operations of the leaf descendants of c.

Lemma 26 *Let* c *be reachable. Then,* $c \rightarrow Op = \biguplus(opSeq(c))$.

Proof We proceed by induction.
Induction Basis Let c be leaf reachable. Then, $opSeq(c) = [c \rightarrow Op]$. Our Lemma holds, since $\biguplus([op]) = op$, by definition.
Induction Step Let c be non-leaf reachable. Suppose that $(c \rightarrow LC \neq \bot) \wedge (c \rightarrow RC \neq \bot)$. (Other cases are similar.)

$$\begin{aligned}
c \rightarrow Op &= (c \rightarrow LC \rightarrow Op) \bigotimes (c \rightarrow RC \rightarrow Op) \\
&= \biguplus(opSeq(c \rightarrow LC)) \bigotimes \biguplus(opSeq(c \rightarrow RC)) \\
&= \biguplus(opSeq(c \rightarrow LC) \circ opSeq(c \rightarrow RC)) \\
&= \biguplus(opSeq(c))
\end{aligned}$$

□

Lemma 27 is the immediate consequence of Lemma 26, which says that $c \rightarrow Op$ is the operation that results from combining, using$\bigotimes$, the operations of the leaf descendants of c.

Lemma 27 *Let* s *be any state. Let* c *be reachable. Then,* $\delta_{state}(s, c \rightarrow Op) = \delta^*_{state}(s, opSeq(c))$.

Proof This is an immediate corollary of Lemma 26, and the observation that $\delta_{state}(s, \biguplus(\sigma)) = \delta^*_{state}(s, \sigma)$.

□

3.5.2.6 *State* Field of a Cell

Lemma 28 says that any non-$\bot$ value in $c \rightarrow State$ is the state that results from applying the operations in $priorOpSeq(c)$ to the implemented object in the initial state. Furthermore, $c \rightarrow State$ is stable, *i.e.* once it holds a non-$\bot$ value, such a value will never change.

Lemma 28 *Let* c *be reachable.*
(a) If $c \rightarrow State \neq \bot$, *then*

$c \rightarrow State = \delta^*_{state}(\text{INITIALSTATE}, priorOpSeq(c))$.

(b) Let $c \rightarrow State = d$, $d \neq \perp$, *at time* t. *Then at any time* t' *such that* $t' \geq t$, $c \rightarrow State = d$.

Proof
Induction Basis Let c be root reachable. Let $List[1] = [r_0, r_1, r_2, \cdots]$. Let $c = r_i, i \geq 1$. $r_i \rightarrow State$ can be assigned a non-$\perp$ value only in Line 6 of `promote`(1). From Line 3, $head = r_{i-1}$. Thus, the non-$\perp$ value of $r_i \rightarrow State$ is $\delta_{state}(r_{i-1} \rightarrow State, r_{i-1} \rightarrow Op)$. Since $r_0 \rightarrow State = anchor_1 \rightarrow State =$INITIALSTATE, $r_0 \rightarrow Op = \perp$, then by a simple induction on i, we have:

If $c \rightarrow State = r_i \rightarrow State \neq \perp$, then

$$c \rightarrow State = \delta^*_{state}(\text{INITIALSTATE}, [r_1 \rightarrow Op, r_2 \rightarrow Op, \cdots, r_{i-1} \rightarrow Op])$$

Recall that

$$leafSeq(\mathcal{R}) = leafSeq(r_1) \circ leafSeq(r_2) \circ \cdots leafSeq(r_{i-1}) \circ leafSeq(r_i) \circ \cdots$$

Thus, $priorOpSeq(c) = opSeq(r_1) \circ opSeq(r_2) \circ \cdots opSeq(r_{i-1})$. Therefore,

$$\begin{aligned}
c \rightarrow State &= \delta^*_{state}(\delta_{state}(\text{INITIALSTATE}, r_1 \rightarrow Op), [r_2 \rightarrow Op, r_3 \rightarrow Op, \cdots, r_{i-1} \rightarrow Op]) \\
&= \delta^*_{state}(\delta^*_{state}(\text{INITIALSTATE}, opSeq(r_1)), [r_2 \rightarrow Op, r_3 \rightarrow Op, \cdots, r_{i-1} \rightarrow Op])(\text{Lemma 27}) \\
&= \delta^*_{state}(\delta^*_{state}(\text{INITIALSTATE}, opSeq(r_1) \circ opSeq(r_2)), [r_3 \rightarrow Op, r_4 \rightarrow Op, \cdots, r_{i-1} \rightarrow Op]) \\
&= \delta^*_{state}(\text{INITIALSTATE}, opSeq(r_1) \circ opSeq(r_2)) \circ \cdots \circ opSeq(r_{i-1})) \\
&= \delta^*_{state}(\text{INITIALSTATE}, priorOpSeq(c))
\end{aligned}$$

This proves part (a). It is easy to see that $r_0 \rightarrow State = anchor_1 \rightarrow State =$INITIALSTATE is stable. By a simple induction on the index i, we see that $\forall i$, $r_i \rightarrow State$ is also stable. Thus, part (b) holds.
Induction Step Let c be non-root reachable.

The Induction Hypothesis is that Lemma 28 holds for $c \rightarrow Parent$. $c \rightarrow State$ can be assigned a non-$\perp$ value only in Line 5 or 6 of `percolateState`, and only if $p \rightarrow State \neq \perp$, where $p = c \rightarrow Parent$. By Lemma 21, when a process is executing `percolateState`, $p = c \rightarrow Parent$ implies that $p = parent(c)$ *i.e.* $p \rightarrow LC = c$ or $p \rightarrow RC = c$.
Case 1 $p \rightarrow LC = c$.

In this Case, $priorOpSeq(c) = priorOpSeq(p)$. By Line 5 of `percolateState`, if $c \rightarrow State \neq \perp$, then

$$\begin{aligned}
c \rightarrow State &= p \rightarrow State \\
&= \delta^*_{state}(\text{INITIALSTATE}, priorOpSeq(p)) \quad \text{(Induction Hypothesis)} \\
&= \delta^*_{state}(\text{INITIALSTATE}, priorOpSeq(c))
\end{aligned}$$

Case 2 $p \rightarrow RC = c$.

In this Case, $priorOpSeq(c) = priorOpSeq(p) \circ opSeq(p \rightarrow LC)$. Since $lop = lc \rightarrow Op$ in `announce`, we have $p \rightarrow Lop = p \rightarrow LC \rightarrow Op$. By Line 6 of `percolateState`, if $c \rightarrow State \neq \perp$, then

$$\begin{aligned}
c \rightarrow State &= \delta_{state}(p \rightarrow State, p \rightarrow Lop) \\
&= \delta_{state}(\delta^*_{state}(\text{INITIALSTATE}, priorOpSeq(p)), p \rightarrow Lop) \quad \text{(Induction Hypothesis)} \\
&= \delta_{state}(\delta^*_{state}(\text{INITIALSTATE}, priorOpSeq(p)), p \rightarrow LC \rightarrow Op) \\
&= \delta^*_{state}(\delta^*_{state}(\text{INITIALSTATE}, priorOpSeq(p)), opSeq(p \rightarrow LC)) \quad \text{(Lemma27)} \\
&= \delta^*_{state}(\text{INITIALSTATE}, priorOpSeq(p) \circ opSeq(p \rightarrow LC)) \\
&= \delta^*_{state}(\text{INITIALSTATE}, priorOpSeq(c))
\end{aligned}$$

This completes the proof of part (a). By Induction Hypothesis, $c \rightarrow Parent \rightarrow State$ is stable. It follows immediately that $c \rightarrow State$ is also stable. This proves part (b).

□

Lemma 29 says that when a root reachable cell has been head, its *State* field holds a non-$\perp$ value.

Lemma 29 *Let* $List[1] = [r_0, r_1, \cdots]$. $\forall i, i \geq 0$, *(*r_i *has been head)*$\Rightarrow$ *(*$r_i \rightarrow State \neq \perp$*).*

Proof

Induction Basis $i = 0$.

At initialization, $r_0 = anchor_1$ is head, and $r_0 \rightarrow State \neq \perp$. Thus, our Lemma holds.

Induction Step Suppose Lemma 29 holds for r_{i-1}.

We now prove that (r_i has been head)$\Rightarrow$ ($r_i \rightarrow State \neq \perp$). If r_i has been head, then some process P has executed a successful SC($Head[1], r_i$) in Line 7 of `promote`(1). When P executes Line 6 , $newcell = r_i$, and $head = r_{i-1}$ (Lines 2,3). Thus, r_{i-1} has been head when P executes Line 2. By the Induction Hypothesis, $r_{i-1} \rightarrow State \neq \perp$. As a result, $r_i \rightarrow State = \delta_{state}(r_{i-1} \rightarrow State, r_{i-1} \rightarrow Op) \neq \perp$ (Line 6). This completes the proof.

□

Lemma 30 says that after `percolateState`(*opcell*, *n*+*P*) terminates, *opcell*'s *State* field holds a non-$\perp$ value. By Lemma 28, we therefore have the following: after `percolateState`(*opcell*, *n*+*P*) terminates, $opcell \rightarrow State$ holds the state that results from applying the operations in *priorOpSeq*(*opcell*) (which is the sequence of operations of the leaf reachable cells to the left of *opcell*) to the implemented object in the initial state.

Lemma 30 *After* `percolateState`*(opcell,* n+P*) in Line 4 of* `apply` *terminates,* $opcell \rightarrow State \neq \perp$*.*

Proof By Lemma 21, before process P begins `percolateState`, $\forall i, 0 \le i \le \log n - 1, a_i \rightarrow Parent = a_{i+1}$, where $a_0 = opcell, a_{i+1} = parent(a_i), a_{\log n}$ is root reachable. This implies that eventually P makes the recursive call `percolateState`$(a_{\log n}, 1)$. Further, by Lemma 21, $a_{\log n}$ has been head.

By Lemma 29, $a_{\log n} \rightarrow State \neq \bot$. In Lines 5,6 of `percolateState`, $(p \rightarrow State \neq \bot) \Rightarrow (c \rightarrow State \neq \bot)$, *i.e.* $(a_{i+1} \rightarrow State \neq \bot) \Rightarrow (a_i \rightarrow State \neq \bot)$. Hence, $a_{\log n} \rightarrow State \neq \bot$ implies that $\forall i, 0 \le i \le \log n - 1$, $a_i \rightarrow State \neq \bot$ when `percolateState`(*opcell*, n+P) terminates. In particular, $opcell \rightarrow State = a_0 \rightarrow State \neq \bot$.

□

3.5.2.7 Linearizability

Lemma 31(c) asserts that `apply`$(P, op, \mathcal{O})$ returns a response that is the same as the response that `apply`$(P, op, \mathcal{O})$ would have obtained if the invocations of the run $\mathcal{R}$ were applied sequentially, in the order specified in the invocation sequence, $invocSeq(\mathcal{R})$. This is a crucial result, and a short step from finally proving linearizability. Theorem 1 provides the full argument for linearizability.

Lemma 31 *Let* $\mathcal{R}$ *be a run. Let* $leafSeq(\mathcal{R}) = [c_1, c_2, \cdots]$. *Then,*
(a) $invocSeq(\mathcal{R}) = [\texttt{apply}(*, c_1 \rightarrow Op, \mathcal{O}), \texttt{apply}(*, c_2 \rightarrow Op, \mathcal{O}), \cdots]$,
(b) $\forall i, i = 1, 2, \cdots :$ *after Line 4 of* `apply`$(*, c_i \rightarrow Op, \mathcal{O})$,
$c_i \rightarrow State = \delta^*_{state}(\text{INITIALSTATE}, [c_1 \rightarrow Op, c_2 \rightarrow Op, \cdots, c_{i-1} \rightarrow Op])$,
(c) $\forall i, i = 1, 2, \cdots :$ `apply`$(*, c_i \rightarrow Op, \mathcal{O})$ *returns*
$\delta_{resp}(\delta^*_{state}(\text{INITIALSTATE}, [c_1 \rightarrow Op, c_2 \rightarrow Op, \cdots, c_{i-1} \rightarrow Op]), c_i \rightarrow Op)$.

Proof
(a) Let $invocSeq(\mathcal{R}) = [\texttt{apply}(*, op_1, \mathcal{O}), \texttt{apply}(*, op_2, \mathcal{O}), \cdots]$. Thus, c_i=*leaf* $(\texttt{apply}(*, op_i, \mathcal{O}))$. By Lemma 22, $op_i = c_i \rightarrow Op$.
(b) $opSeq(\mathcal{R}) = [c_1 \rightarrow Op, c_2 \rightarrow Op, \cdots, c_{i-1} \rightarrow Op, c_i \rightarrow Op, \cdots]$. Therefore, $priorOpSeq(c_i) = [c_1 \rightarrow Op, c_2 \rightarrow Op, \cdots, c_{i-1} \rightarrow Op]$. By Lemmas 30 and 28, after Line 4 of `apply`$(*, c_i \rightarrow Op, \mathcal{O})$,

$$\begin{aligned} c_i \rightarrow State &= \delta^*_{state}(\text{INITIALSTATE}, priorOpSeq(c_i)) \\ &= \delta^*_{state}(\text{INITIALSTATE}, [c_1 \rightarrow Op, c_2 \rightarrow Op, \cdots, c_{i-1} \rightarrow Op]) \end{aligned}$$

(c) This follows immediately from part (b) of this Lemma.

□

3.5.3 Summary

Theorem 1 *The closed object construction shown in Figures 3.6 to 3.9 is linearizable and wait-free.*

Proof The closed object construction is wait-free, by inspection of the algorithm.

Consider a run $\mathcal{R}$. By Lemmas 23, 24 and 25, *invocSeq*($\mathcal{R}$) satisfies the first three requirements of Definition 7.

Consider a run $\mathcal{R}'$ in which the invocations in *invocSeq*($\mathcal{R}$) are applied sequentially to the implemented object, initialized to INITIALSTATE. Let procedure `apply`$(P, op, \mathcal{O})$ return x in $\mathcal{R}$. Then, by Lemma 31(a),(c), `apply`$(P, op, \mathcal{O})$ returns x in $\mathcal{R}'$. Thus, *invocSeq*($\mathcal{R}$) satisfies the last requirement of Definition 7. Therefore $\mathcal{R}$ is linearizable. Since $\mathcal{R}$ is an arbitrary run, our construction is linearizable.

□

Theorem 2 *The shared-access time complexity and local time complexity of the closed object construction shown in Figures 3.6 to 3.9 are both* $O(\log^2 n)$.

Proof

We make the following observations:

1. A call to `readyOrphan`, `announce`, or `combine` results in $O(1)$ accesses to shared objects and $O(1)$ local steps.
2. A call to `append`(*lst*) results in $O(1)$ shared-memory steps and $O(1)$ local steps.
3. When the recursion in the `promote` procedure is eliminated, a call to `promote` results in at most $2 * \log n$ calls to `append`, thus giving rise to $O(\log n)$ shared-memory steps and $O(\log n)$ local steps.
4. A call to `percolateState`(c, *lst*), or `release`(c, *lst*, $*$) results in $O(\log n)$ shared-memory steps and $O(\log n)$ local steps.
5. `apply` makes one call to `announce`, $1 + \log n$ calls to `promote`, one call to `percolateState`, and one call to `release`. Therefore, a call to `apply` results in $O(\log^2 n)$ shared-memory steps and $O(\log^2 n)$ local steps.

Thus, both the shared-access time complexity and local time complexity of the closed object construction are $O(\log^2 n)$.

□

Chapter 4

Bounded Construction for Closed Objects

4.1 General Principles

In the construction presented in Chapter 3, each time a process P invokes either `announce` or `combine`, it allocates a new cell from its private pool of cells. Thus, each process requires a pool of unbounded number of cells. We now describe the general approach that we have taken in modifying the unbounded construction, in order to bound the total number of cells required.

First, we recall that once a cell c becomes head, any further changes to the fields of c come in one of two ways:

- processes that installed the leaf descendants of c write to $c \rightarrow State$ during their execution of `percolateState`.
- processes try to write to $c \rightarrow Next$ during their execution of `combine` or `announce`.

We provide every cell c with two additional fields: $c \rightarrow Free$ and $c \rightarrow Retired$, to indicate whether such changes to the fields of c may still happen.

$c \rightarrow Free = true$ indicates that all the leaf descendants of c have completed their `percolateState`. Thus, c will no longer be accessed by any process executing `percolateState`. $c \rightarrow Retired = true$ indicates that c has been head, but is no longer head. Thus, $c \rightarrow Next$ points to some cell d, and d has been head. We say that a cell c is *invalid* if both $c \rightarrow Free$ and $c \rightarrow Retired$ hold *true*. The fields of an invalid cell no longer holds any useful information. Our aim is to *recycle* invalid cells. A recycled cell is said to be in a new *incarnation*.

There is no guarantee that an invalid cell c is no longer accessed by processes. Since $c \rightarrow Free = true$ means that all the leaf descendants of c have completed their `percolateState`, we know that no process will access an invalid c during `percolateState`. However, it is possible, for example, for a process P to access an invalid c during Line 2 of `readyOrphan`: Suppose P reads $head := c$ in Line 1 when c is head. If P waits until c is invalid before executing Line 2, then P accesses an invalid c.

Therefore, we need to ensure that, whenever an invalid c is accessed, the algorithm behaves in exactly the same way, whether c has been recycled (so that c is in a new incarnation, and its fields hold indeterminate values) or not. In other words, the contents of an invalid c must not affect the correctness of the algorithm. Furthermore, if c has been recycled, then the contents of c must be protected from any change by the processes that intend to access an old incarnation of c.

To achieve this, we first observe the following: If a cell c is accessed by a process P in procedures other than `percolateState`, then the following is true:

P first reads $Head[lst]$, for some lst, and sets $head := Head[lst]$. P then follows some pointers, starting from $head$, to reach c: Either $head = c, head \rightarrow Next = c, head \rightarrow Next \rightarrow LC = c$, or $head \rightarrow Next \rightarrow RC = c$.

Suppose that initially P had executed $head :=$ LL($Head[lst]$), instead of $head := Head[lst]$. Further suppose that P executes VL($Head[lst]$) at some time t when c is invalid. We can prove that at t, $Head[lst]$ no longer equals $head$. Thus, the VL operation returns *false*. Hence, if P executes VL($Head[lst]$) immediately after reading values from c, and discards such values if VL($Head[lst]$) returns *false*, then values from an invalid cell c will never be used by P. If, on the other hand, VL($Head[lst]$) returns *true*, then the values read are indeed from a valid c. In this case, P proceeds with the steps of the construction in Chapter 3.

We therefore require that every *read* operation on c be preceded by a LL($Head[lst]$), and followed by a VL($Head[lst]$). If the VL operation returns *false*, then P abandons any further operation on c, as no useful work remains to be done on c. On the other hand, if the VL operation returns *true*, then P is assured that it has read values from a valid c. In this case, it proceeds with the normal execution. By this mechanism, we ensure that the contents of an invalid c (which are indeterminate) do not affect the correctness of our algorithm.

After reading from a valid c, P may want to write to c. We require that any *write* to c observes certain rules. For example, consider the case where P wants to write x to $c \rightarrow Parent$. We require that P first executes $parent :=$LL($c \rightarrow Parent$). P then executes VL($Head[lst]$), to be sure that the value $parent$ indeed came from a valid c. P then checks to see if $parent = \perp$. If $parent \neq \perp$, P does nothing. If $parent = \perp$, P executes SC($c \rightarrow Parent, x$).

We note that when P executes SC($c \rightarrow Parent, x$), c may be invalid. If c is invalid, then the SC operation must fail. This is true for the following reason: Before P executes the SC operation, P has executed the LL operation when c is valid, and $c \rightarrow Parent = \perp$. $c \rightarrow Parent$ must have held a non-$\perp$ value, before c becomes invalid. Hence, when P executes the SC operation (at a time when c is invalid), the SC operation must fail.

This ensures that the contents of an invalid c are not changed by P. Thus, if c has been recycled, the contents of the current incarnation of c are not corrupted.

In summary, we use VL($Head[lst]$) to ensure that either the values read from c are from a valid c, or further operations on c are abandoned. Furthermore, we ensure that there is no successful *write* operations on an invalid c. With these mechanisms, an invalid cell can be safely recycled.

As we shall prove, our bounded implementation requires only $3n(3n + \log n)$ cells.

4.2 Bounded Space Complexity* (BSC*) Implementation

We incorporate the changes outlined in the previous section into the unbounded construction presented in Chapter 3 to obtain the algorithm (which we call the Bounded Space Complexity* (BSC*) implementation) in Figures 4.1 to 4.5. BSC* differs from the final bounded implementation that we desire in only one particular: In BSC*, `announce` and `combine` always return *new* cells. Thus, with BSC*, even though invalid cells can be safely recycled, no attempt is made to recycle them. At a later section, we will present the final bounded implementation, which we call the Bounded Space Complexity (BSC) implementation. In BSC, `announce` and `combine` return either a new cell or a recycled, invalid cell. Thus, invalid cells are recycled in BSC.

The purpose of BSC* is to serve as a convenient tool in proving the correctness of BSC.

4.2.1 `release` procedure

The only major difference between BSC* and the unbounded construction in Chapter 3 is the addition of the procedure `release`. We explain below how it works.

Process P executes `release`($opcell, n + P$, *left*) (Line 6 of `apply`) after it has completed `percolateState`. Thus, P executes `release` after it has written the appropriate value into $opcell \rightarrow State$. The purpose of `release` is to signal to the cells along the path from *opcell* to its root ancestor w that P has completed its `percolateState`.

Recall that each cell c has a field $c \rightarrow Free$. $c \rightarrow Free = true$ indicates that all the leaf descendants of c have completed their `percolateState`. In addition to $c \rightarrow Free$, c has two more fields: $c \rightarrow LDone$ and $c \rightarrow RDone$. $c \rightarrow LDone = true$ indicates that either $c \rightarrow LC = \perp$ or all the leaf descendants of $c \rightarrow LC$ have completed their `percolateState`. Likewise, $c \rightarrow RDone = true$ indicates that either $c \rightarrow RC = \perp$ or all the leaf descendants of $c \rightarrow RC$ have completed their `percolateState`.

For any cell c, if $c \rightarrow LC \neq \perp$, then there is exactly one leaf descendant l of $c \rightarrow LC$ that *represents* $c \rightarrow LC$. We also say that the process that installed l *represents* $c \rightarrow LC$. The process that represents $c \rightarrow LC$ is responsible for setting $c \rightarrow LDone$ to *true*. Similarly, if $c \rightarrow RC \neq \perp$, then there is exactly one leaf descendant r of $c \rightarrow RC$ that *represents* $c \rightarrow RC$. The process that installed r *represents* $c \rightarrow RC$, and is responsible for setting $c \rightarrow RDone$ to *true*.

Suppose that d is an ancestor of P's *opcell*, and P represents $d \rightarrow LC$. In this case, P will execute Line 4 of `release`. After P sets $d \rightarrow LDone$ to *true* (Line 4), it checks $d \rightarrow RDone$ (Line 5). If $d \rightarrow RDone = true$, then P executes SC($d \rightarrow Free, true$) (Line 6). If the SC operation succeeds, then P represents d, and P executes `ascend`. In `ascend`, if d is a left child of e, then P represents $e \rightarrow LC$, and makes a recursive call `release`($e, *,$*left*). Likewise, if d is a right child of e, then P represents $e \rightarrow RC$, and makes a recursive call `release`($e, *, right$). Thus, the unique process that executes a successful SC($d \rightarrow Free, true$) represents d to d's parent e.

If after P sets $d \rightarrow LDone$ to *true* (Line 4), it finds that $d \rightarrow RDone =$*false*, then P returns from `release`. Likewise, if P executes a SC($d \rightarrow Free, true$) that returns *false*, then then P returns from `release`. Thus, P begins at *opcell* at the leaf position, and proceeds up the path from *opcell* to the root w, as far as P is able to represent the cells along the path.

We observe the following: If P has not begun `release`, then for all cells d along the path from *opcell* to the root w, $d \rightarrow Free =$*false*. Furthermore, let c be any cell. if $c \rightarrow Free =$*true*, then

all the leaf descendants of c have completed their `percolateState`.

4.3 Proof of Correctness of BSC*

Our first objective is to prove that BSC* is linearizable, and wait-free. More specifically, we want to show that Theorems 1 and 2 in Chapter 3 hold with respect to BSC*. Our proofs of these Theorems follow exactly the structure of their proofs in Chapter 3. However, the many changes made to the algorithm in Chapter 3 (in particular, the insertion of many instructions of the form **if not** VL(*) **return**), require that the proofs be carefully examined, to verify that the arguments still hold.

We re-prove here Lemmas 1 to 17 with respect to BSC*. The proofs are essentially the same as those presented in Chapter 3, with certain non-trivial differences. In particular, the proofs of Lemmas 11, 14, 16, and 17 with respect to BSC* show significant difference from their proofs in Chapter 3.

The proofs in Chapter 3 of Lemmas 18 to 31, and Theorems 1 and 2 (with obvious, and minor changes) are easily seen as applicable to BSC*. We therefore omit re-proving them here. Since all the Lemmas in Chapter 3 hold with respect to BSC*, we conclude, by Theorem 1 (with respect to BSC*), that BSC* is linearizable and wait-free.

Lemma 1 *Let c be a cell returned either by* `announce` *to Line 1 of* `apply`*, or by* `combine` *to Line 7 of* `append`*.*
(a) The value of $c \rightarrow LC$ is unchanged.
(b) The value of $c \rightarrow RC$ is unchanged.
(c) The value of $c \rightarrow Lop$ is unchanged.
(d) The value of $c \rightarrow Op$ is unchanged.
(e) Let $c \rightarrow Next = d$, $d \neq \perp$, at time t. Then at any time t' such that $t' \geq t$, $c \rightarrow Next = d$.
(f) Let $c \rightarrow Parent = d$, $d \neq \perp$, at time t. Then at any time t' such that $t' \geq t$, $c \rightarrow Parent = d$.
(g) Let $c \rightarrow Ready = true$, at time t. Then at any time t' such that $t' \geq t$, $c \rightarrow Ready = true$.

Proof
(a),(b) $c \rightarrow LC$ and $c \rightarrow RC$ are assigned values only once: in Line 1 of `announce` or Line 1 of `combine`. This observation implies Lemmas 1(a) and 1(b).
(c),(d) $c \rightarrow Lop$ and $c \rightarrow Op$ are assigned values only once: in Line 1 of `announce` or Line 1 of `combine`. This observation implies Lemmas 1(c) and 1(d).
(e)We first note that $c \rightarrow Next$ is assigned the value $\perp$, only during the initialization of c in `announce` and `combine`, and nowhere else. There are only two places where $c \rightarrow Next$ can take on a non-$\perp$ value: Line 4 of `announce`, and Line 9 of `append`. If process P executes a successful SC($c \rightarrow Next, d$), $d \neq \perp$, in Line 4 of `announce`, then P must have previously executed LL($c \rightarrow Next$) in Line 3, with $\perp$ as the LL operation's return value. This implies Lemma 1(e).

If process P executes a successful SC($c \rightarrow Next, d$), $d \neq \perp$, in Line 9 of `append`, then P must have previously executed LL($c \rightarrow Next$) in Line 2, with $\perp$ as the LL operation's return value (Line 3). This implies Lemma 1(e).
(f)We first note that $c \rightarrow Parent$ is assigned the value $\perp$, only during the initialization of c in `announce` and `combine`, and nowhere else. There are only two places where $c \rightarrow Parent$ can take

on a non-$\perp$ value: Lines 21, 35 of `append`. If process P executes a successful SC($c \rightarrow Parent, d$), $d \neq \perp$, in Line 21 (resp. 35) of `append`, then P must have previously executed LL($c \rightarrow Parent$) in Line 20 (resp. 34), with $\perp$ as the LL operation's return value. This implies Lemma 1(f).
(g)This follows from the fact that $c \rightarrow Ready$ is assigned *false* only during the initialization of c in `combine`, and nowhere else.

□

Lemma 2 *Let c be any cell. There is at most one cell b such that $b \rightarrow Next = c$.*

Proof There are only two places where $b \rightarrow Next$ (for some b) can be assigned the value c: Line 4 of `announce`, Line 9 of `append`. In both of these places, process P initializes a new cell c and executes SC($b \rightarrow Next, c$). Given any cell c, at most one process executes the operation SC($b \rightarrow Next, c$) (for some b); further, this process executes the operation once only. Hence, there is at most one cell b such that $b \rightarrow Next = c$.

□

Lemma 3 *Let lst, lst' be such that $1 \leq lst, lst' \leq 2n-1$. Let i, j be such that $i \geq 0, j \geq 0$. Then, $(List[lst](i) = List[lst'](j)) \Rightarrow ((lst = lst') \wedge (i = j))$.*

Proof This Lemma follows from Lemma 2.

□

Lemma 4 *Let $k \geq 0$. Suppose at some time t $Head[lst] = List[lst'](k)$.[1] Let c be the first value that is successfully written to $Head[lst]$ after t. Let t' be the time c is successfully written to $Head[lst]$. Then,*

- *$c = List[lst'](k+1)$.*
- *At time τ such that $\tau \geq t'$, $List[lst'](k) \rightarrow Next = List[lst'](k+1)$.*

Proof $Head[lst]$ can be written to in three places: Line 11 of `promote`, Lines 27,41 of `append`. In all these places, $Head[lst]$ changes value from b, the value of $Head[lst]$ returned by the immediately preceding LL($Head[lst]$) (Line 2 of `promote`, Lines 22,36 of `append`), to $c = b \rightarrow Next$ through a successful SC($Head[lst], c$). Further, by Line 4 of `promote`, Lines 17,26 of `append`, Lines 31,40 of `append`, we have $b \rightarrow Next = c \neq \perp$. By Lemma 1(c), once $b \rightarrow Next = c$ holds, $b \rightarrow Next = c$ at all subsequent times. This implies our Lemma.

□

[1]Lemma 5(a) shows that $lst = lst'$.

```
    initialization
    ∀i, 1 ≤ i ≤ 2n-1 : anchor_i : *CELL

1.    for i := 1 to 2n-1:
2.       anchor_i → Next := ⊥, anchor_i → Free := true,
         anchor_i → Retired := false, Head[i] := anchor_i
3.    anchor_1 → State := INITIALSTATE
4.    anchor_1 → Op := ⊥
    end initialization

    apply (P : INTEGER, op : OP, 𝒪) returns RES
1.    opcell := announce (op, P)
2.    for i := 0 to log n
3.       promote (⌊(n+P)/2^i⌋)
4.    percolateState (opcell, n+P)
5.    release (opcell, n+P, left)
6.    return δ_resp (opcell → State, op)
    end apply


    promote (lst : INTEGER)
1.    if lst = 1
2.       head := LL(Head[lst])
3.       newcell := head → Next
4.       if newcell = ⊥ return
5.       if newcell → Ready = false return
6.       newstate := δ_state (head → State, head → Op)
7.       newcellstate := LL(newcell → State)
8.       if not VL(Head[lst]) return
9.       if newcellstate = ⊥
10.         SC(newcell → State, newstate)
11.      if SC(Head[lst], newcell)
12.         head → Retired := true
13.      return
14.   append (⌊lst/2⌋)
15.   promote (⌊lst/2⌋)
16.   append (⌊lst/2⌋)
    end promote
```

Figure 4.1: BSC* construction for closed object $\mathcal{O}$ (Figures 4.1 to 4.5)

```
    append (lst : INTEGER)
1.    head := LL(Head[lst])
2.    newcell := LL(head → Next)
3.    if not VL(Head[lst]) return
4.    if newcell = ⊥
5.       (lc, lop) := readyOrphan (2 ∗ lst)
6.       (rc, rop) := readyOrphan (2 ∗ lst + 1)
7.       if (lc ≠ ⊥) or (rc ≠ ⊥)
8.          c := combine (lc, lop, rc, rop)
9.          result := SC (head → Next, c)
10.         if not result
11.            c → Free := true, c → Retired := true
12.      newcell := head → Next
13.      if not VL(Head[lst]) return
14.      if newcell = ⊥ return

15.   lchild := newcell → LC
16.   if not VL(Head[lst]) return
17.   if lchild ≠ ⊥
18.      lcparent := LL(lchild → Parent)
19.      if not VL(Head[lst]) return
20.      if lcparent = ⊥
21.         SC(lchild → Parent, newcell)
22.      lchead := LL(Head[2*lst])
23.      lcnewcell := lchead → Next
24.      if not VL(Head[2*lst]) go to L
25.      if not VL(Head[lst]) return
26.      if lcnewcell = lchild
27.         if SC(Head[2*lst], lchild)
28.            lchead → Retired := true
    (continue on next Figure)
```

Figure 4.2: BSC* construction for closed object $\mathcal{O}$ (Figures 4.1 to 4.5)

```
append (lst : INTEGER)
    (continued from previous Figure)

29.L   rchild := newcell → RC
30.    if not VL(Head[lst]) return
31.    if rchild ≠ ⊥
32.       rcparent := LL(rchild → Parent)
33.       if not VL(Head[lst]) return
34.       if rcparent = ⊥
35.          SC(rchild → Parent, newcell)
36.       rchead := LL(Head[2*lst+1])
37.       rcnewcell := rchead → Next
38.       if not VL(Head[2*lst+1]) go to M
39.       if not VL(Head[lst]) return
40.       if rcnewcell = rchild
41.          if SC(Head[2*lst+1], rchild)
42.             rchead → Retired := true

43.M   ready := LL(newcell → Ready)
44.    if not VL(Head[lst]) return
45.    if ready = false
46.       SC(newcell → Ready, true)
    end append
```

Figure 4.3: BSC* construction for closed object $\mathcal{O}$ (Figures 4.1 to 4.5)

```
   readyOrphan (lst : INTEGER) returns (*CELL, OP)
1.   head := LL(Head[lst])
2.   newcell := head → Next
3.   if newcell = ⊥  return (⊥, ⊥)
4.   if newcell → Ready = false return (⊥, ⊥)
5.   newop := newcell → Op
6.   if not VL(Head[lst]) return (⊥, ⊥)
7.   return (newcell, newop)
   end readyOrphan

   announce (op : OP, P : INTEGER) returns *CELL
1.   allocate a new cell c and initialize it as follows:
       c → Parent := ⊥, c → Next := ⊥, c → State := ⊥
       c → LC := ⊥, c → RC := ⊥,
       c → Op := op, c → Lop := ⊥, c → Ready := true
       c → LDone := false, c → RDone := true, c → Free := false, c → Retired := false
2.   head := Head[n+P]
3.   if LL(head → Next) = ⊥
4.     SC(head → Next, c)
5.   return c
   end announce

   combine (lc : *CELL, lop : OP, rc : *CELL, rop : OP ) returns *CELL
1.   allocate a new cell c and initialize it as follows:
       c → Parent := ⊥, c → Next := ⊥, c → State := ⊥
       c → LC := lc, c → RC := rc,
       c → Lop := lop, c → Op := (lop ⊗ rop)
       c → Ready := false, c → Free := false, c → Retired := false
       if lc ≠ ⊥
         c → LDone := false
       else c → LDone:= true
       if rc ≠ ⊥
         c → RDone := false
       else c → RDone := true
2.   return c
   end combine
```

Figure 4.4: BSC* construction for closed object $\mathcal{O}$ (Figures 4.1 to 4.5)

Lemma 5
(a) The initial value of $Head[lst]$ is $List[lst](0)$. The sequence of values that have been successfully written to $Head[lst]$ at time t is:
$List[lst](1), List[lst](2), \cdots, List[lst](k)$, for some $k \geq 0$.
(b) ($List[lst](i)$ has been head)$\wedge$ ($List[lst](i)$ is not head) $\Rightarrow$ ($List[lst](i+1)$ has been head).
(c) Let $k \geq 1$. At the time $List[lst](k)$ is successfully written to $Head[lst]$, $List[lst](k)$ is reachable.
(d) (c is head at lst)$\Rightarrow$ ($c = anchor_{lst}$) $\vee$ (c is reachable at lst).
(e) Let $c = Head[lst] \rightarrow Next$, then c is reachable at lst.
(f) (c is head at lst) $\wedge$ ($c \rightarrow Next = \perp$) $\wedge$ (d is reachable at lst) $\Rightarrow$ (d has been head at lst)

Proof
(a) We note that at initialization, $Head[lst] = anchor_{lst} = List[lst](0)$. This statement is a corollary of Lemma 4.
(b) This follows immediately from Part(a).
(c) The proof proceeds by induction on k, and uses Lemma 4. The Induction Step proceeds as follows: Suppose that $List[lst](k-1)$ is reachable (Induction Hypothesis). Then, at the time $List[lst](k)$ is successfully written to $Head[lst]$, $List[lst](k-1) \rightarrow Next = List[lst](k)$ (by Lemma 4). Thus, by Definition 1, $List[lst](k)$ is reachable.
(d) This is a re-statement of Part(c).
(e) We note that $Head[lst]$ is head at lst. By Part(d), we have ($Head[lst] = anchor_{lst}$)$\vee$($Head[lst]$ is reachable at lst). In either case, $c = Head[lst] \rightarrow Next$ is reachable at lst.
(f) By Part(a), there exists a k such that $c = List[lst](k)$. Since ($c \rightarrow Next = \perp$)$\wedge$($d$ is reachable at lst), we have $d = List[lst](m)$, for some m such that $k \geq m \geq 1$. By Part(a), d has been head at lst.

□

Lemma 6 *Let $b \rightarrow Next = c$. (c is reachable at lst) $\Rightarrow$ (b has been head at lst).*

Proof From the algorithm, $b \rightarrow Next$ is assigned the value c in either Line 4 of `announce`, or Line 9 of `append`. In both these places, a process first executes $head := Head[lst']$ for some lst' (Line 2 of `announce`, Line 1 of `append`), and then changes the value of $head \rightarrow Next$ from $\perp$ to c through a successful SC($head \rightarrow Next, c$). After the successful SC($head \rightarrow Next, c$), $Head[lst'] \rightarrow Next = c$. By Lemma 5(e), c is reachable at lst'. Hence, $lst' = lst$. Since $Head[lst] \rightarrow Next = c$ and $b \rightarrow Next = c$, $Head[lst] = b$ (Lemma 2). In other words, b has been head at lst. This completes our proof.

□

Lemma 7 *Let c be non-leaf reachable.*
(a) $c \rightarrow LC = d \neq \perp \Rightarrow$ (d is reachable) $\wedge$($pos(d) = 2 * pos(c)$)$\wedge$ (d is ready).
(b) $c \rightarrow RC = d \neq \perp \Rightarrow$ (d is reachable) $\wedge$($pos(d) = 2 * pos(c) + 1$)$\wedge$ (d is ready).

Proof (a) $c \rightarrow LC$ is assigned the value d in `combine` (d, *, *, *), called in Line 8 of `append` (lst). Further, $(d, *)$ is the value returned by `readyOrphan` ($2 * lst$) (Line 5 of `append` (lst)). In `readyOrphan` ($2 * lst$), $head = Head[2 * lst]$ (Line 1). $d = head \rightarrow Next$ (Line 2), $d \rightarrow Ready =$

```
    percolateState (c : *CELL, lst : INTEGER)
1.    if lst = 1 return
2.    p := c → Parent
3.    percolateState (p, ⌊lst/2⌋)
4.    if lst = 2 ∗ ⌊lst/2⌋
5.       c → State := p → State
6.    else c → State := δ_state (p → State, p → Lop)
    end percolateState
```

```
    release (c : *CELL, lst : INTEGER, dir : {left, right})
1.    p := c → Parent
2.    LL(c → Free)
3.    if dir = left
4.       c → LDone := true
5.       if c → RDone
6.          if SC(c → Free, true)
7.             ascend (p, lst)
8.    else c → RDone := true
9.       if c → LDone
10.         if SC(c → Free, true)
11.            ascend (p, lst)
    end release
```

```
    ascend (c : *CELL, lst : INTEGER)
1.    if lst =1 return
2.    if lst = 2 ∗ ⌊lst/2⌋
3.       release (c, ⌊lst/2⌋, left)
4.    else release (c, ⌊lst/2⌋, right)
    end ascend
```

Figure 4.5: BSC* construction for closed object $\mathcal{O}$ (Figures 4.1 to 4.5)

true (Line 4). Therefore, d is reachable at $2 * lst$ (by Lemma 5(e)). Since c is reachable, some process P executed a successful SC($head \to Next, c$) in Line 9 of `append`. When P executed Line 1 of `append`, $head = Head[lst]$. Therefore, $c = head \to Next$ is reachable at lst (by Lemma 5(e)). Thus, $pos(c) = lst$, and $pos(d) = 2 * lst = 2 * pos(c)$. Finally, d is ready, since $d \to Ready = true$.
(b) Similar to the proof of part (a).

□

Lemma 8 *Let c be non-leaf reachable. $c = parent(d) \Rightarrow$ (d is reachable) $\wedge$(d is ready).*

Proof This follows immediately from Lemma 7.

□

Lemma 9 *Let d be non-root reachable. (d has been head)$\Rightarrow \exists$ reachable c such that $c = parent(d)$.*

Proof d becomes head in either Line 27 or Line 41 of `append`. Consider the case of Line 27. (The case of Line 41 is analogous.) Let P be the process that executed a successful SC($Head[2*lst], d$), where $d = lchild$, in Line 27, thus causing d to become head. We note that during P's execution of `append`, $lchild = newcell \to LC$ (Line 15), $newcell = head \to Next$ (Line 2 or 12), $head = Head[lst]$ (Line 1). Let c in our Lemma be *newcell*. Then, by Lemma 5(e), c is reachable. By Notation 3, $c = parent(d)$ since $c \to LC = lchild = d$. This completes the proof.

□

Lemma 10 *Let d be such that $\forall lst, 1 \leq lst \leq 2n-1, d \neq anchor_{lst}$. ($d$ has been head) $\Rightarrow$ (d is ready).*

Proof Since $\forall lst, 1 \leq lst \leq 2n-1, d \neq anchor_{lst}$, and d has been head, d is reachable (by Lemma 5(d)). If d is non-root reachable, Lemma 10 is an immediate corollary of Lemmas 8 and 9.

Consider the case in which d is root reachable. d becomes $Head[1]$ through a successful SC($Head[1], d$) in Line 11 of `promote` (1). Further, $d \to Ready = true$ (Line 5). Hence, d is ready.

□

Lemma 11 *Let c be non-leaf reachable. Then, (c is ready)$\wedge$($c = parent(d)$) $\Rightarrow$ (d has been head).*

Proof Consider the case of $c \to LC = d$. (The case of $c \to RC = d$ is analogous.) c becomes ready through process P executing a successful SC($c \to Ready, true$) in Line 46 of `append`. Further, since $lchild = c \to LC = d \neq \perp$ (Lines 15,17), P executes Line 22 before executing Line 46. At the time t when P executes Line 22, $c = parent(d)$ holds true. By Lemma 8, ($c = parent(d)$) $\Rightarrow$ (d is reachable). Let $d = List[pos(d)](k)$. Let $x = List[pos(d)](k-1)$. By Lemma 6, x has been head at $pos(d)$ at time t.

If x is not head at t, then, by Lemma 5(b), d has been head at t. Therefore our Lemma holds in this case. Suppose x is head at t. In Line 23 of `append`, $lcnewcell = x \to Next = d$.

In Line 24, if VL($Head[pos(d)]$) returns *false*, then d has been head (by Lemma 5(b)) when P executes Line 24. The proof is done in this case. Suppose the VL operation in Line 24 returns *true*. Since $lchild = d$ (Line 15), P observes that $lcnewcell = lchild$ in Line 26, and executes

SC($Head[pos(d)], d$) in Line 27. If the SC operation returns *true*, then d has been head when P completes Line 27, hence also when P completes Line 46. If the SC operation returns *false*, then by Lemma 5(a), d has been head when P begins executing Line 27. In either case, d has been head when P executes Line 46. This completes the proof. □

Lemma 12 *Let c be non-leaf reachable. (c has been head)$\wedge(c = parent(d)) \Rightarrow$ (d has been head).*

Proof By Lemma 10, (c has been head) $\Rightarrow$ (c is ready). By Lemma 11, (c is ready)$\wedge(c = parent(d)) \Rightarrow$ (d has been head). Thus, our Lemma holds.

□

Lemma 13 *Let b, c be non-leaf reachable. $(b = parent(d)) \wedge (c = parent(d)) \Rightarrow (b = c)$.*

Proof For contradiction, assume $b \neq c$. By Lemma 7, $pos(b) = pos(c)$. Let $b = List[pos(b)](k), c = List[pos(b)](l)$. Without loss of generality, let $k < l$. Let P be the process that executes a successful SC($head \rightarrow Next, c$) in Line 9 of `append`. Let t be the time P executes Line 1 of this `append`. Thus, at t, $Head[pos(b)] = List[pos(b)](l-1)$. Since $k < l$, b has been head at t. By Lemma 12, since $b = parent(d)$, d has been head at t.

$c = parent(d)$ implies that when P executes `readyOrphan` ($pos(d)$) in Line 5 or 6 of `append` at some time after t , $(d, *)$ is returned. This in turn implies that when P executes `readyOrphan` ($pos(d)$), $head = Head[pos(d)]$ in Line 1, and $head \rightarrow Next = d$ in Line 2. In other words, d has not been head when P executes Line 1 of `readyOrphan` ($pos(d)$). Thus, d has not been head at t. This contradiction proves that $b = c$.

□

Lemma 14 *Let d be non-root reachable. (d has been head)$\Rightarrow \exists$ unique non-leaf reachable c such that $(c = parent(d)) \wedge (d \rightarrow Parent = c)$.*

Proof By Lemmas 9, (d has been head)$\Rightarrow \exists$ reachable c such that $c = parent(d)$. We now show that c is non-leaf. By Lemma 7, since d is non-root reachable, and $pos(d) = 2 * pos(c)$, we conclude that c is non-leaf reachable. By Lemma 13, such a c is unique. It remains to prove that (d has been head)$\Rightarrow (d \rightarrow Parent = c)$.

We consider the case of $c \rightarrow LC = d$. (The case of $c \rightarrow RC = d$ is analogous.) Suppose that d becomes head by process P executing a successful SC($Head[pos(d)], d$) in Line 27 of `append`. This implies that $lchild = d$ (Line 26). In Line 15, $lchild = newcell \rightarrow LC$. By Lemma 13, $newcell = c$. We note that P executes Line 20 before executing Line 27 (or 41, in the case of $c \rightarrow RC = d$). Further, $lcparent = lchild \rightarrow Parent = d \rightarrow Parent$ in Line 20. If $lcparent = \perp$ (Line 20), then P executes SC($d \rightarrow Parent, c$) (Line 21). If the SC operation returns *true*, then $d \rightarrow Parent = c$ after P executes Line 21. Our Lemma is thus proved.

However, if either $d \rightarrow Parent \neq \perp$ (Line 20) or the SC operation in Line 21 returns *false*, then $d \rightarrow Parent \neq \perp$ before P executes Line 22. $d \rightarrow Parent$ is assigned a non-$\perp$ value only through some process P' executing a successful SC($d \rightarrow Parent, newcell$) in Line 21 of `append`. However, when P' executes Line 15, $d = newcell \rightarrow LC$. By Lemma 13, $newcell = c$. Therefore, the non-$\perp$ value assigned by P' to $d \rightarrow Parent$ is c. This completes the proof. □

Lemma 15 *Let $lst > 1$. Suppose at time t, $Head[lst] = b$, $Head[\lfloor lst/2 \rfloor] = c$. Then, ($b$ has been head)$\wedge$(b is not head)$\Rightarrow$ ($c \to Next \neq \bot$).*

Proof For contradiction, suppose that at time t', (b has been head)$\wedge$ (b is not head)$\wedge$($c \to Next = \bot$). By Lemma 5(a), $t' > t$. By Lemma 5(b), there exists a reachable d such that $b \to Next = d$, and d has been head at t'. By Lemma 9, there exists a reachable a such that $a = parent(d)$ at t'. Since c is head, $c \to Next = \bot$, and a is reachable at $pos(c) = \lfloor lst/2 \rfloor$ at t', we conclude that a has been head at t' (Lemma 5(f)). Since c is head at both t and t', a has been head at t. By Lemma 12, (a has been head)$\wedge$($a = parent(d)$) $\Rightarrow$ (d has been head) at t. This contradicts the assumption that b is head at t. Our Lemma is thus proved by this contradiction. □

Lemma 16 *Suppose that at time t before process P invokes* `append`*($\lfloor lst/2 \rfloor$), $Head[lst] = b, b \to Next = d \neq \bot, d$ is ready, $Head[\lfloor lst/2 \rfloor] = c$. Then, at any time after* `append`*($\lfloor lst/2 \rfloor$) terminates, $\exists$ reachable e such that ($c \to Next = e$) $\wedge$ (e is ready).*

Proof We note that c is head at t. If at any time during the execution of `append`($\lfloor lst/2 \rfloor$) c is not head, then by Lemma 5(b), $\exists$ reachable e such that ($c \to Next = e$), and e has been head. By Lemma 10, e is ready. Our Lemma is then proved.

We now consider the case where c is head throughout the execution of `append`($\lfloor lst/2 \rfloor$). This implies that VL($Head[\lfloor lst/2 \rfloor]$) in Lines 3,13,16,19,25,30,33,39,44 always return *true*. Consider process P executing `append`($\lfloor lst/2 \rfloor$). In Line 1, $head = Head[\lfloor lst/2 \rfloor] = c$.

Case 1 Suppose $\exists$ reachable e such that $c \to Next = e \neq \bot$ in Line 2.

Since VL($Head[\lfloor lst/2 \rfloor]$) always returns *true*, Line 45 is always executed. $newcell = e$ (Line 2), $ready = e \to Ready$ (Line 43). We note that for any non-leaf reachable g, the only place $g \to Ready$ is changed, after g's initialization in `combine`, is a successful SC operation in Line 46 of `append`. This implies that, at the time P completes Lines 45,46, $e \to Ready = true$. Hence our Lemma holds.

Case 2 Suppose $c \to Next = \bot$ in Line 2.

Consider P executing `readyOrphan` (lst) (Line 5 or 6 of `append`). Let t' be the time P executes Line 1 of `readyOrphan` (lst).

Case 2a Suppose $Head[lst] \neq b$ at t'.

By Lemma 15, since (b has been head)$\wedge$ (b is not head) at t', we have $c \to Next \neq \bot$ at t'. Therefore, there exists a reachable e such that $newcell = c \to Next = e \neq \bot$ in Line 12 of `append`. This implies that P always executes Line 45. Further, $ready = e \to Ready$ ((Line 43). By the argument used in Case 1, at the time P completes Lines 45,46, $e \to Ready = true$. Hence, our Lemma holds.

Case 2b Suppose $Head[lst] = b$ at t'.

By the premise of our Lemma, in Lines 1-4 of `readyOrphan` (lst), $newcell = b \to Next = d \neq \bot, newcell \to Ready = true$. If VL($Head[lst]$) returns *false* in Line 6 of `readyOrphan` (lst), then (b has been head)$\wedge$(b is not head) when P executes Line 6. By an analogous argument to the one used in Case 2a, our Lemma holds. If VL($Head[lst]$) returns *true* in Line 6, then `readyOrphan` (lst) returns $(b, *)$. P then executes SC($c \to Next, *$) in Line 9 of `append`. Whether the SC operation returns *true* or *false*, $c \to Next \neq \bot$ when P executes Line 12. $newcell = c \to Next \neq \bot$ (Line 12). By the argument used in Case 2a, our Lemma holds in this case, too. □

Lemma 17 *Let r be root reachable and ready before the invocation of* `promote`*(1). Then, after* `promote`*(1) terminates, r has been head.*

Proof Let s be root reachable or $anchor_1$, such that $s \rightarrow Next = r$. Thus, if $r = List[1](k)$, then $s = List[1](k-1)$. By Lemma 6, since r is reachable, s has been head before process P invokes `promote`(1). Suppose that at some time t during P's execution of `promote`(1), $Head[1] \neq s$. By Lemma 5(b), r has been head at t. Thus, our Lemma holds.

We now consider the case where $Head[1] = s$ throughout the execution of `promote`(1). This implies that $head = Head[1] = s$ and VL($Head[1]$) in Line 8 returns *true*. Since $newcell = s \rightarrow Next = r \neq \perp$ (Line 4), $newcell \rightarrow Ready = r \rightarrow Ready = true$ (Line 5), P executes SC($Head[1], r$) in Line 11. Whether the SC operation returns *true* or *false*, $Head[1] \neq s$ after P's execution of Line 11 (Lemma 4). This contradicts the assumption that $Head[1] = s$ throughout the execution of `promote`(1). The proof is now complete. □

As we explained earlier, Lemmas 18 to 31 hold with respect to BSC* as well. Consequently, we have the following Theorems, which are the counterparts of Theorems 1 and 2 with respect to the BSC* construction:

Theorem 3 *The BSC* construction shown in Figures 4.1 to 4.5 is linearizable and wait-free.*

Theorem 4 *The shared-access time complexity and local time complexity of the BSC* construction shown in Figures 4.1 to 4.5 are both $O(\log^2 n)$.*

4.4 Properties of BSC*

In this section, we prove certain properties of BSC* that will be useful in proving the correctness of the BSC implementation.

4.4.1 Invalid Cells

We use two fields, $Free$ and *Retired*, in a cell c to indicate whether c is available for recycling. We first define an *invalid* cell as a cell whose $Free$ and *Retired* fields are both *true*. Thus, an invalid cell is ready to be recycled.

Definition 17 *A reachable cell c is* invalid *if and only if* $c \rightarrow Free = \text{true} \wedge c \rightarrow Retired = \text{true}$. *$c$ is* valid *if and only if c is not invalid.*

The next Lemma says that once $Free$ and *Retired* hold the value *true*, they hold that value forever. Hence, once a cell is invalid, it remains invalid forever.

Lemma 32 *Let c be reachable.*
(a) ($c \rightarrow Free = true$ at time t)$\Rightarrow$ $(\forall t', t' \geq t : c \rightarrow Free = true$ at time $t')$.
(b) ($c \rightarrow Retired = true$ at time t)$\Rightarrow$ $(\forall t', t' \geq t : c \rightarrow Retired = true$ at time $t')$.
(b) (c is invalid at time t)$\Rightarrow$ $(\forall t', t' \geq t : c$ is invalid at time $t')$.

Proof Let c be any cell. The only times when $c \rightarrow Free := false$ or $c \rightarrow Retired := false$ are executed are during initialization (when $anchor_i$ is initialized), or when c is initialized in `announce` or `combine`. Hence, once $c \rightarrow Free =$ true, or $c \rightarrow Retired =$ true, holds, it holds forever. Therefore, our Lemma holds. □

4.4.2 Properties of $c \rightarrow Free$

Recall that $c \rightarrow Free = true$ indicates that all the leaf descendants of c have completed their `percolateState`. In this section, we prove various useful properties of $c \rightarrow Free$.

Lemma 33

- *In* `release`$(c, pos(c), *)$, *if* c *is a left child of* $c \rightarrow Parent$, *then the next recursive call to* `release` *(if made) is* `release`$(c \rightarrow Parent, pos(c \rightarrow Parent), left)$.
- *In* `release`$(c, pos(c), *)$, *if* c *is a right child of* $c \rightarrow Parent$, *then the next recursive call to* `release` *(if made) is* `release`$(c \rightarrow Parent, pos(c \rightarrow Parent), right)$.

Proof This follows immediately from Lines 2,3,4 of `ascend`. □

Lemma 34 $(c \rightarrow Free) \Rightarrow (c \rightarrow LDone) \wedge (c \rightarrow RDone)$

Proof $c \rightarrow Free$ is set to $true$ in either Line 6 or 10 of `release`. This Lemma is a consequence of Lines 4-6, 8-10 of `release`. □

Lemma 35 *Let* c *be non-leaf reachable.*

- *If* $c \rightarrow LC = \perp$, *then* $c \rightarrow LDone = true$, *and* `release`$(c, pos(c), left)$ *is never invoked.*
- *If* $c \rightarrow RC = \perp$, *then* $c \rightarrow RDone = true$, *and* `release`$(c, pos(c), right)$ *is never invoked.*

Proof If $c \rightarrow LC = \perp$, then by Line 1 of `combine`, $c \rightarrow LDone = true$. In Line 5 of `apply`, `release`$(d, pos(d), left)$, where d is leaf reachable, is invoked by process P. All further `release`'s executed by P are recursively invoked within this `release`$(d, pos(d), left)$. By Lemma 33, if `release`$(c, pos(c), left)$ is ever invoked, then $c \rightarrow LC \neq \perp$. Hence, `release`$(c, pos(c), left)$ is never invoked.

The case of $c \rightarrow RC = \perp$ is analogous. □

Lemma 36 *Let* c *be non-leaf reachable. Suppose that the following statements hold:*

- $(c \rightarrow LC \neq \perp) \Rightarrow \exists$a unique invocation of `release`$(c, pos(c), left)$.
- $(c \rightarrow RC \neq \perp) \Rightarrow \exists$a unique invocation of `release`$(c, pos(c), right)$.

Then, the following statements hold:
(a) In any invocation of `release`$(c, pos(c), *)$, c *is valid when Line 4 or 8 is executed.*
(b) Let $c \rightarrow Free = true$ *at time* t. *Any execution of SC(*$c \rightarrow Free, true$*) after* t *returns* false.
(c) $c \rightarrow Free = true$ *eventually.*
(d)

- *If c is a left child, then there exists a unique invocation of* `release`*($c \to Parent, pos(c \to Parent), left$).*

- *If c is a right child, then there exists a unique invocation of* `release`*($c \to Parent, pos(c \to Parent), right$).*

(e) Let $p = c \to Parent$. Then,

$$(p \to Free) \Rightarrow (p \to LC = \perp \vee p \to LC \to Free) \wedge (p \to RC = \perp \vee p \to RC \to Free).$$

Proof We observe the following:

- $c \to LC \neq \perp \vee c \to RC \neq \perp$ (Line 7 of `append`).

- If $c \to LC = \perp$, then $c \to LDone = true$, and no `release`$(c, pos(c), left)$ is invoked (Lemma 35).

- If $c \to LC \neq \perp$, then exactly one `release`$(c, pos(c), left)$ is invoked (premise of this Lemma). Further, $c \to LDone = false$ before any `release`$(c, pos(c), left)$ is invoked.

- If $c \to RC = \perp$, then $c \to RDone = true$, and no `release`$(c, pos(c), right)$ is invoked (Lemma 35).

- If $c \to RC \neq \perp$, then exactly one `release`$(c, pos(c), right)$ is invoked (premise of this Lemma). Further, $c \to RDone = false$ before any `release`$(c, pos(c), right)$ is invoked.

(a) We assume that $c \to LC \neq \perp$, and $c \to RC \neq \perp$. (It is easy to see how our proof can be adapted in the other cases.) $c \to LDone = c \to RDone = false$ on c's initialization. $c \to LDone$ ($c \to RDone$ resp.) can become $true$ only when Line 4 (Line 8 resp.) of `release`$(c, pos(c), left)$ (`release`$(c, pos(c), right)$ resp.) is executed. Let P (P' resp.) be the process that invokes the `release`$(c, pos(c), left)$ (`release`$(c, pos(c), right)$ resp.). By Lemma 34, since $c \to LDone = false$ immediately before P executes Line 4, $c \to Free = false$ when P executes Line 4. By Definition 17, c is valid when P executes Line 4. Similarly, $c \to Free = false$ when P' executes Line 8. By Definition 17, c is valid when P' executes Line 8. This proves Lemma 36(a).
(b) Since $c \to Free = false$ when P executes Line 4, therefore $c \to Free = false$ when P executes Line 2. Consequently, P executes LL($c \to Free$) in Line 2 before any SC($c \to Free, true$) is executed. Likewise P' executes LL($c \to Free$) in Line 2 before any SC($c \to Free, true$) is executed. Only P, P' may execute SC($c \to Free, true$) (Lines 6 and 10). Therefore, the first SC operation (if one exists) returns *true*, and the second SC operation (if one exists) returns *false*. This proves Lemma 36(b).
(c) If $c \to RDone = true$ when P executes Line 5, then $c \to Free = true$ when P completes the SC operation in Line 6. If $c \to RDone = false$ when P executes Line 5, then P' will subsequently execute Line 8, setting $c \to RDone$ to *true*. Then P' will successfully execute SC($c \to Free, true$) in Line 10. Hence, $c \to Free = true$ eventually. This proves Lemma 36(c).
(d) Let c be a left child. The unique process, P or P', that executes a successful SC($c \to Free, true$) proceeds to call `release`$(c \to Parent, pos(c \to Parent), left)$. No other process will

ever invoke release$(c \rightarrow Parent, pos(c \rightarrow Parent), left)$. An analogous argument holds for the case when c is a right child. This completes the proof of Lemma 36(d).
(e) Let $p = c \rightarrow Parent$. Suppose $p \rightarrow LC = c$. $p \rightarrow Free = true$ implies that $p \rightarrow LDone = true$. This in turn implies that release$(p, pos(p), left)$ has been invoked. This further implies that $c \rightarrow Free = true$. This proves Lemma 36(e).

□

Lemma 37 *Let c be leaf reachable. Then,*
(a) In any invocation of release$(c, pos(c), *)$*, c is valid when Line 4 or 8 is executed.*
(b) Let $c \rightarrow Free = true$ *at time t. Any execution of SC(*$c \rightarrow Free, true$*) after t returns* false*.*
(c) $c \rightarrow Free = true$ *eventually.*
(d)

- *If c is a left child, then there exists a unique invocation of* release$(c \rightarrow Parent, pos(c \rightarrow Parent), left)$*.*
- *If c is a right child, then there exists a unique invocation of* release$(c \rightarrow Parent, pos(c \rightarrow Parent), right)$*.*

(e) Let $p = c \rightarrow Parent$*. Then,*

$$(p \rightarrow Free) \Rightarrow (p \rightarrow LC = \bot \vee p \rightarrow LC \rightarrow Free) \wedge (p \rightarrow RC = \bot \vee p \rightarrow RC \rightarrow Free).$$

Proof $c \rightarrow RDone = true, c \rightarrow LDone = false$, during c's initialization in announce. There exists exactly one invocation of release$(c, pos(c), left)$, called in Line 5 of apply. This is analogous to the case in Lemma 36, where c is non-leaf reachable, with $c \rightarrow LC \neq \bot$, and $c \rightarrow RC = \bot$. The proof of Lemma 36 applies here too. □

Lemma 38 *Let c be non-leaf reachable. Then,*

- $(c \rightarrow LC \neq \bot) \Rightarrow \exists$ *unique invocation of* release$(c, pos(c), left)$*.*
- $(c \rightarrow RC \neq \bot) \Rightarrow \exists$ *unique invocation of* release$(c, pos(c), right)$*.*

Proof Define $level(c) = l$ such that $2^l \leq pos(c) < 2^{l+1}$. Thus, a root cell is at level 0, and a leaf cell is at level $\log n$. Let $level(c) = l$. We prove inductively that Lemma 38 holds for all descendants of c at level $\log n - 1, \log n - 2, \cdots, l$.
Induction Basis Applying Lemma 37(d) to the leaf descendants (at level $\log n$) of c, Lemma 38 holds for all descendants of c at level $\log n - 1$.
Induction Step Applying Lemma 36(d) to the descendants of c at level k, Lemma 38 holds for all descendants of c at level $k - 1$.

□

Lemma 39 *Let c be reachable. Then, the following statements hold:*
(a) In any invocation of release$(c, pos(c), *)$*, c is valid when Line 4 or 8 is executed.*
(b) Let $c \rightarrow Free = true$ *at time t. Any execution of SC(*$c \rightarrow Free, true$*) after t returns* false*.*

(c) $c \rightarrow Free = true$ *eventually.*
(d) Let $p = c \rightarrow Parent$*. Then,*

$$(p \rightarrow Free) \Rightarrow (p \rightarrow LC = \perp \vee p \rightarrow LC \rightarrow Free) \wedge (p \rightarrow RC = \perp \vee p \rightarrow RC \rightarrow Free).$$

Proof If c is leaf reachable, then Lemma 37 holds. Lemma 37(a), (b), (c), (e) are identical with Lemma 39(a), (b), (c), (d).

If c is non-leaf reachable, then Lemma 38 implies that Lemma 36 is applicable. Lemma 36(a), (b), (c), (e) are identical with Lemma 39(a), (b), (c), (d).

□

Lemma 40 *Let* g *be a leaf descendant of* c*. Then,* $c \rightarrow Free \Rightarrow g \rightarrow Free$*.*

Proof By Lemma 39(d), for any non-leaf reachable c, if $c \rightarrow Free$, then $c \rightarrow LC = \perp \vee c \rightarrow LC \rightarrow Free$, $c \rightarrow RC = \perp \vee c \rightarrow RC \rightarrow Free$. Repeated application of Lemma 39(d) yields our Lemma. □

Lemma 41 *Let* c *be reachable, and be accessed at time* t *by a process executing* `percolateState`*. Then* c *is valid at* t*.*

Proof Let reachable c be accessed by some process P executing `percolateState`$(opcell, n+P)$. Then, c is an ancestor of *opcell*, *i.e.* *opcell* is a leaf descendant of c. Let I be the interval during which P is executing `percolateState`$(opcell, n+P)$. Since $opcell \rightarrow Free$ can become $true$ only when P executes `release`$(opcell, n+P, left)$, $opcell \rightarrow Free = false$ throughout the interval I. By Lemma 40, $c \rightarrow Free = false$ throughout I. Thus, c is valid throughout I. Our Lemma is therefore proved. □

Lemma 42 *Let* c *be reachable.*

$$(c \rightarrow Free = true) \Rightarrow (c \rightarrow Parent \text{ has been head}).$$

Proof Let c be reachable. $c \rightarrow Free = false$ during the initialization of c in `announce` of `combine`. The only places where $c \rightarrow Free$ becomes $true$ are Lines 6 and 10 of `release`.

Suppose P executes SC$(c \rightarrow Free, true)$ in Line 6 or 10 of `release`. We note that when P executes `release`, P has completed the **for** loop in Lines 2 to 3 of `apply`. By Lemma 21, if $p = c \rightarrow Parent$, then $p = parent(c)$, *i.e.* $(p \rightarrow LC = c) \vee (p \rightarrow RC = c)$. Thus, c is an ancestor of *opcell*. (By Definition 5, this includes the possibility that $c = opcell$.) $c \rightarrow Parent$ is therefore also an ancestor of *opcell*. By lemma 21, every ancestor of *opcell* has been head. Therefore, $c \rightarrow Parent$ has been head. □

4.4.3 Properties of $c \rightarrow Retired$

Recall that $c \rightarrow Retired = true$ indicates that c has been head, but is no longer head. Thus, $c \rightarrow Next$ points to some cell d, and d has been head. We prove various useful properties of $c \rightarrow Retired$ in this section.

Lemma 43 *Let c be reachable.*

$$(c \rightarrow Retired = true) \Rightarrow (c \text{ has been head}) \wedge (c \text{ is not head})$$

Proof Let c be reachable. $c \rightarrow Retired$ =*false* during the initialization of c in `announce` or `combine`. The only places where $c \rightarrow Retired$ becomes *true* are Line 12 of `promote`, Lines 28 and 42 of `append`. In all these places, process P executes $c \rightarrow Retired := true$ only if P has first read $Head[lst] = c$, and then executed a successful SC($Head[lst], c \rightarrow Next$). (Specifically, P reads $Head[lst]$ in Line 2 of `promote`, Lines 22 and 36 of `append`. P executes the successful SC in Line 11 of `promote`, Lines 27 and 41 of `append`.) Hence, if $c \rightarrow Retired = true$, then ($c$ has been head)$\wedge$(c is not head). □

Lemma 44 *Let c be a reachable cell. If (c has been head)$\wedge$(c is not head), then:*
(a) there is exactly one execution of $c \rightarrow Retired := true$*, and no execution of SC(*$c \rightarrow Retired, *$*),*
(b) $c \rightarrow Retired = true$ *eventually,*

Proof During c's initialization, $c \rightarrow Retired$ =*false*. If P executes $c \rightarrow Retired := true$ in Line 11 of `append`, then c is not reachable. Thus, the only places where, for a reachable c, $c \rightarrow Retired := true$ is executed are: Line 12 of `promote`, Lines 28 and 42 of `append`. In each case, P first executes $head$:=LL($Head[l]$) (Line 2 of `promote`, Lines 22 and 36 of `append`). P then executes a successful SC($Head[l], head \rightarrow Next$) (Line 11 of `promote`, Lines 27 and 41 of `append`). Finally, P executes $head \rightarrow Retired := true$ (Line 12 of `promote`, Lines 28 and 42 of `append`).

Since (c has been head)$\wedge$(c is not head), there has been a successful SC($Head[*], c \rightarrow Next$). Thus, there is exactly one execution of $head \rightarrow Retired = true$. Lemma 44(a), (b) therefore follow. □

4.4.4 Preliminary Lemmas

The Lemmas in this section concern the behavior of a VL or SC operation on an invalid cell c, given certain conditions that relate the VL or SC operation to the immediately preceding LL operation on c. These results are the foundation for the Lemmas on reading from, and writing to, invalid cells in the next two sections.

Lemma 45 *Suppose process P first executes headcell :=LL(*$Head[lst]$*), then executes VL(*$Head[lst]$*) at a later time t. Let c be either* $headcell, headcell \rightarrow Next, headcell \rightarrow Next \rightarrow LC$*, or* $headcell \rightarrow Next \rightarrow RC$*. If c is invalid at some time before t, then VL(*$Head[lst]$*) at t returns* false.

Proof By Lemma 32, since c is invalid at some time before t, c is invalid at t.
(a) Let $c = headcell$. Suppose that VL($Head[lst]$) at t returns *true*. This implies that $Head[lst] = headcell = c$ at t. Thus, c is head at t. As observed, c is invalid at t. Therefore, $c \rightarrow Retired = true$ at t. By Lemma 43, c is not head at t. This contradiction completes the proof.
(b) Let $c = headcell \rightarrow Next$. Suppose that VL($Head[lst]$) at t returns *true*. This implies that $Head[lst] = headcell$ at t. By Lemma 5(a), c has not been head at t. As observed, c is invalid at

t. Therefore, $c \to Retired = true$ at t. By Lemma 43, c has been head at t. This contradiction completes the proof.
(c) Let $c = headcell \to Next \to LC$. Suppose that VL($Head[lst]$) at t returns $true$. This implies that $Head[lst] = headcell$ at t. Let $d = headcell \to Next$. Thus, $d \to LC = c$. By Notation 3, $d = parent(c)$. d has not been head at t. c is invalid at t. Thus, $(c \to Retired = true) \wedge (c \to Free = true)$ at t. By Lemma 43, c has been head at t. By Lemma 14, $c \to Parent = d$ at t. By Lemma 42, $c \to Parent = d$ has been head at t. This contradicts the conclusion that d has not been head at t. This completes the proof.
(d) Let $c = headcell \to Next \to RC$. The proof is analogous to the proof of part (c).

□

Lemma 46 *Let c be reachable. Suppose P executes LL($c \to Next$) when c is valid, and the LL operation returns $\perp$. If P next accesses $c \to Next$ by executing SC($c \to Next, *$) when c is invalid, then the SC operation returns* false.

Proof This Lemma is an immediate consequence of the following Claims:
(a) If $c \to Next = d \neq \perp$ at time t, then $\forall t' \geq t, c \to Next = d$ at time t'.
(b) (c is invalid)$\Rightarrow c \to Next \neq \perp$.

Claim (a) is Lemma 1(a). We now prove Claim (b). Suppose c is invalid. By definition, $c \to Retired = true$. By Lemma 43, (c has been head)$\wedge$(c is not head). By Lemma 5(b), 5(d), $c \to Next \neq \perp$. Thus, Claim (b) holds. □

Lemma 47 *Let c be reachable. Suppose P executes LL($c \to Parent$) when c is valid, and the LL operation returns $\perp$. If P next accesses $c \to Parent$ by executing SC($c \to Parent, *$) when c is invalid, then the SC operation returns* false.

Proof This Lemma is an immediate consequence of the following Claims:
(a) If $c \to Parent = d \neq \perp$ at time t, then $\forall t' \geq t, c \to Parent = d$ at time t'.
(b) (c is invalid)$\Rightarrow c \to Parent \neq \perp$.

Claim (a) is Lemma 1(d). We now prove Claim (b). Suppose c is invalid. By Lemma 43, c has been head. By Lemma 14, $c \to Parent \neq \perp$. Thus, Claim (b) holds. □

Lemma 48 *Let c be reachable. Suppose P executes LL($c \to Ready$) when c is valid, and the LL operation returns* false. *If P next accesses $c \to Ready$ by executing SC($c \to Ready, *$) when c is invalid, then the SC operation returns* false.

Proof This Lemma is an immediate consequence of the following Claims:
(a) If $c \to Ready = true$ at time t, then $\forall t' \geq t, c \to Ready = true$ at time t'.
(b) (c is invalid)$\Rightarrow c \to Ready = true$.

Claim (a) is Lemma 1(e). We now prove Claim (b). Suppose c is invalid. By Lemma 43, (c has been head). By Lemma 10, c is ready. Thus, Claim (b) holds. □

Lemma 49 *Let c be root reachable. Suppose P executes LL($c \to State$) when c is valid, and the LL operation returns $\perp$. If P next accesses $c \to State$ by executing SC($c \to State, *$) when c is invalid, then the SC operation returns* false.

Proof This Lemma is an immediate consequence of the following Claims:
(a) If $c \rightarrow State = d \neq \perp$ at time t, then $\forall t' \geq t, c \rightarrow State = d$ at time t'.
(b) (c is invalid)$\Rightarrow c \rightarrow State \neq \perp$.

We now prove Claim (a). Let c be root reachable. The only place where $c \rightarrow State$ can take on a non-$\perp$ value is in Line 10 of `promote`. If process P executes a successful SC$(c \rightarrow State, d), d \neq \perp$, in Line 10 of of `promote`, then P must have previously executed LL$(c \rightarrow State)$ in Line 7, with $\perp$ as the LL operation's return value. This proves Claim (a).

We now prove Claim (b). Suppose c is invalid. By Lemma 43, c has been head. By Lemma 29, $c \rightarrow State \neq \perp$. Thus, Claim (b) holds. □

4.4.5 Reading from Invalid Cells

In this section, our aim is to show that if all invalid cells hold indeterminate values (*i.e.* all useful values that such a cell held when it was valid have been lost), the BSC* implementation would still function correctly. This clearly means that process P has to know whether the values that it read came from a valid or invalid cell. If such values came from an invalid cell, then they are unreliable, and P must discard them.

In the Lemmas that follow, we show, procedure by procedure, that all the component procedures (including `apply` would still function correctly, even if all invalid cells hold indeterminate values. This is a crucial property of BSC* that permits recycling of invalid cells.

Lemma 50 *Suppose the following condition holds:*

> $\forall$ *process* $P, \forall c, t$ *such that* c *is reachable and invalid at* t*: if while* P *is executing* `readyOrphan`*,* P *reads from any field in* c *at* t*, then the value returned to* P *is indeterminate (*i.e. *arbitrary). (If* P *reads from* c *while executing any other procedure, then* c *returns the correct value to* P*.)*

Then, our construction remains correct.

Proof In `readyOrphan`, P executes $head$:=LL$(Head[lst])$ in Line 1, and VL$(Head[lst])$ in Line 6. P reads values from $head$ in Line 2, $newcell = head \rightarrow Next$ in Lines 4 and 5. P does not write to any shared objects (namely, $Head[*]$ and reachable cells) in `readyOrphan`.

Suppose that when P reads from $c = head$ in Line 2, c is invalid. Then, P's `readyOrphan` returns with $(\perp, \perp)$ in either Line 3,4, or 6. (This is because, if P does not return in Line 3 or 4, then P executes VL$(Head[lst])$ in Line 6. By Lemma 45, this VL operation returns *false*. Thus, P returns with $(\perp, \perp)$ in Line 6.) Even though the values in the various fields of c that are returned to P are indeterminate, they do not affect the correctness of our construction.

Suppose now that when P reads from $c = head \rightarrow Next$ in Line 4, c is invalid. Then, P's `readyOrphan` returns with $(\perp, \perp)$ in either Line 4 or 6. (The argument here is analogous to the one used for $c = head$.)

Suppose finally that when P reads from $c = head \rightarrow Next$ in Line 5, c is invalid. Then, P's `readyOrphan` returns with $(\perp, \perp)$ in Line 6.

Our Lemma is thus proved. □

Lemma 51 *Suppose the following condition holds:*

> $\forall$ *process* P , $\forall\, c, t$ *such that* c *is reachable and invalid at* t*: if while* P *is executing* `combine`*,* P *reads from any field in* c *at* t*, then the value returned to* P *is indeterminate (*i.e. *arbitrary). (If* P *reads from* c *while executing any other procedure, then* c *returns the correct value to* P*.)*

Then, our construction remains correct.

Proof This is trivially true, since P reads from no reachable cell in `combine`. □

Lemma 52 *Suppose the following condition holds:*

> $\forall$ *process* P , $\forall\, c, t$ *such that* c *is reachable and invalid at* t*: if while* P *is executing* `append`*,* P *reads from any field in* c *at* t*, then the value returned to* P *is indeterminate (*i.e. *arbitrary). (If* P *reads from* c *while executing any other procedure, then* c *returns the correct value to* P*.)*

Then, our construction remains correct.

Proof The proof is similar in principle to the proof of Lemma 50. We examine the various segments of `append` where reachable cells are read, and prove that the correctness of our construction is not affected by the condition stated in our Lemma.

Segment 1: Lines 1-3

P executes $head := LL(Head[lst])$. P reads from $head$ in Line 2. If $head$ is invalid, P returns at Line 3 (by Lemma 45).

Segment 2: Lines 4-11

Lines 5-6: Lemma 50 applies.

Line 8: Lemma 51 applies.

P does not read from reachable cells in other Lines.

Segment 3: Lines 12-14

P reads from $head$ in Line 12. If $head$ is invalid, P returns at Line 13.

Segment 4: Lines 15-16

P reads from $head \rightarrow Next$ in Line 15. If $head \rightarrow Next$ is invalid, P returns at Line 16.

Segment 5: Lines 17-21

P reads from $head \rightarrow Next \rightarrow LC$ in Line 18. If $head \rightarrow Next \rightarrow LC$ is invalid, P returns at Line 19.

Segment 6: Lines 22-28

P executes $lchead$:=LL($Head[2 * lst]$) in Line 22. P then reads from $lchead$ in Line 23. If $lchead$ is invalid, P goes to Line 29, without making any further use of the value, $lcnewcell$, read from $lchead$ (Line 23).

Segment 7: Lines 29-42

This segment is analogous to Lines 15-28.

Segment 8: Lines 43-46

P reads from $head \rightarrow Next$ in Line 43. If $head \rightarrow Next$ is invalid, then P returns at Line 44. □

Lemma 53 *Suppose the following condition holds:*

> $\forall$ *process* P, $\forall c, t$ *such that* c *is reachable and invalid at* t*: if while* P *is executing* `promote`*,* P *reads from any field in* c *at* t*, then the value returned to* P *is indeterminate (*i.e. *arbitrary). (If* P *reads from* c *while executing any other procedure, then* c *returns the correct value to* P*.)*

Then, our construction remains correct.

Proof
Segment 1: Lines 1-13

P first executes $head$:=LL($Head[lst]$) in Line 1. P reads from $head$ in Line 3. If $head$ is invalid at this point, then P either returns at Line 4, Line 5, or reads from $head$ (Line 6), $head \rightarrow Next$ (Line 7), and then returns at Line 8. In all of these cases, P executes no SC or *write* operation on any shared object.

P reads from $head \rightarrow Next$ in Line 5. If $head \rightarrow Next$ is invalid at this point, then P returns at either Line 5 or Line 8.

P reads from $head$ in Line 6. If $head$ is invalid at this point, then P returns at Line 8.

P reads from $head \rightarrow Next$ in Line 7. If $head \rightarrow Next$ is invalid at this point, then P returns at Line 8.

Segment 2: Lines 14-16

Lines 14, 16: Lemma 52 applies.

Line 15: Since `promote`(lst) recurses to `promote`(1), Segment 1 serves as the basis of an inductive proof.

□

Lemma 54 *Suppose the following condition holds:*

> $\forall$ *process* P, $\forall c, t$ *such that* c *is reachable and invalid at* t*: if while* P *is executing* `announce`*,* P *reads from any field in* c *at* t*, then the value returned to* P *is indeterminate (*i.e. *arbitrary). (If* P *reads from* c *while executing any other procedure, then* c *returns the correct value to* P*.)*

Then, our construction remains correct.

Proof P executes $head := Head[n + P]$ in Line 2. Let $c = head$. P reads from c at time t in Line 3. We now prove that c is valid at t.

A new cell c becomes reachable at position $n+P$ in only one way: when P executes `announce`, from Line 1 of `apply`. By Lemma 19, when this `apply` terminates, c has become head. When P next executes `announce`, c is head when P executes Line 3 of `announce` at t (since at this point $c \rightarrow Next = \perp$). Now suppose c is invalid at t. By Lemma 43, c is not head at t. This contradiction proves that c is valid at t. Since P does not read from any invalid cell when executing `announce`, our Lemma holds. □

Lemma 55 *Suppose the following condition holds:*

> $\forall$ *process* $P, \forall c, t$ *such that* c *is reachable and invalid at* t*: if while* P *is executing* `percolateState`*,* P *reads from any field in* c *at* t*, then the value returned to* P *is indeterminate (*i.e. *arbitrary). (If* P *reads from* c *while executing any other procedure, then* c *returns the correct value to* P*.)*

Then, our construction remains correct.

Proof This is an immediate consequence of Lemma 41. □

Lemma 56 *Suppose the following condition holds:*

> $\forall$ *process* $P, \forall c, t$ *such that* c *is reachable and invalid at* t*: if while* P *is executing* `release`*,* P *reads from any field in* c *at* t*, then the value returned to* P *is indeterminate (*i.e. *arbitrary). (If* P *reads from* c *while executing any other procedure, then* c *returns the correct value to* P*.)*

Then, our construction remains correct.

Proof In `release`$(c, pos(c), *)$, c is the only reachable cell accessed by P. (We do not consider the recursive call to `release` from `ascend`$(p, pos(c))$.) By Lemma 39(a), the only places where P might read an invalid c are Lines 5 and 9. Suppose P reads $c \rightarrow RDone$ (Line 5) at time t, and c is invalid at t. If $c \rightarrow RDone = false$, P returns from `release`. If $c \rightarrow RDone = true$, c executes SC($c \rightarrow Free, true$) in Line 6. Since c is invalid at t, $c \rightarrow Free = true$ at t. By Lemma 39(b), the SC operation in Line 6 returns *false*. Thus, the values in c after t do not affect the correctness of our construction. The remaining case is that, when P reads $c \rightarrow LDone$ (Line 9), c is invalid. An analogous argument suffices here. Our Lemma is therefore proved. □

Lemma 57 *Suppose the following condition holds:*

> $\forall$ *process* $P, \forall c, t$ *such that* c *is reachable and invalid at* t*: if while* P *is executing* `apply`*,* P *reads from any field in* c *at* t*, then the value returned to* P *is indeterminate (*i.e. *arbitrary). (If* P *reads from* c *while executing any other procedure, then* c *returns the correct value to* P*.)*

Then, our construction remains correct.

Proof

Line 1: Lemma 54 applies.

Line 2-3: Lemma 53 applies.

Line 4: Lemma 55 applies.

Line 5: Lemma 56 applies.

Line 6: We note that when P reads $opcell \rightarrow State$ in Line 6, $opcell$ is head. Thus, by Lemma 43, *opcell* is valid.

Hence, our Lemma holds. □

Lemma 58 *Suppose that the values in the invalid reachable cells are indeterminate. Our construction remains correct.*

Proof This Lemma is a concise re-statement of Lemma 57. □

Definition 18 $\forall i, 1 \leq i \leq 2n-1$*:*

- $anchor_i$ *is called an* anchor*.*
- $anchor_i$ *is* invalid *if and only if* $(anchor_i \rightarrow Retired = true) \wedge (anchor_i \rightarrow Free = true)$*.*
- $anchor_i$ *is* valid *if and only if* $anchor_i$ *is not invalid.*

Lemma 59 *Suppose that the values in an invalid anchor are indeterminate. Our construction remains correct.*

Proof No process accesses an anchor from `percolateState` or `release`, or from Line 6 of `apply`. The arguments used to prove Lemma 58 proves this Lemma. □

4.4.6 Writing to Invalid Cells

In this section, our objective is to show that no process will ever write successfully to an invalid cell. (A process may execute a SC operation on an invalid cell. However, such an operation is bound to fail.) Thus, the contents of an invalid cell are protected from being corrupted by the *write* and SC operations of any process.

In the Lemmas that follow, we show, procedure by procedure, that all the component procedures (including `apply`) prevent processes from writing successfully to an invalid cell. This is another crucial property of BSC* that permits recycling of invalid cells.

Lemma 60 $\forall$ *process* P *,* $\forall\, c, t$ *such that* c *is reachable and invalid at* t*: While executing* `announce`*,* P *does not successfully write to any field in* c *at* t*.*

Proof The only place where P writes to a reachable cell is in Line 4. By the same argument as that used in the proof of Lemma 54, *head* is valid at the time when P executes Line 4. Thus our Lemma holds. □

Lemma 61 $\forall$ *process* P *,* $\forall\, c, t$ *such that* c *is reachable and invalid at* t*: While executing* `readyOrphan`*,* P *does not successfully write to any field in* c *at* t*.*

Proof P does not write to any shared object in `readyOrphan`. □

Lemma 62 $\forall$ *process* P *,* $\forall\, c, t$ *such that* c *is reachable and invalid at* t*: While executing* `combine`*,* P *does not successfully write to any field in* c *at* t*.*

Proof P does not write to any shared object in `combine`. □

Lemma 63 *$\forall$ process P, $\forall c, t$ such that c is reachable and invalid at t: While executing* `append`*, P does not successfully write to any field in c at t.*

Proof

Lines 5-6: Lemma 61 applies.

Line 8: Lemma 62 applies.

Line 10: P has executed $newcell$:=LL($head \rightarrow Next$) in Line 2 at time t. In Line 3, P executed VL($Head[lst]$), which returns *true*. By Lemma 45, $head = Head[lst]$ is valid at t. By Line 4, $newcell = \perp$. Lemma 46 is applicable. Thus, if *head* is invalid when P executes SC($head \rightarrow Next, c$) in Line 10, the SC operation returns *false*.

Line 11: Suppose c executes Line 11, writing to cell c. By Line 10 and the fact that the SC operation in Line 10 returns *false*, c is not reachable.

Line 21: P has executed LL($lchild \rightarrow Parent$) in Line 18 at time t. By Line 19, *lchild* is valid at t. By Line 20, the LL operation in Line 18 returns $\perp$. By Lemma 47, if *lchild* is valid when P executes Line 21, then the SC operation returns *false*.

Line 27: P writes to $Head[2 * lst]$, which is not a cell.

Line 28: P writes to $lchild \rightarrow Retired$. By Lemma 44(a), $lchead \rightarrow Retired$ =*false* when P begins Line 28. Thus, *lchead* is valid when P begins Line 28. □

Lines 35,41,42: similar to Lines 21,27, 28.

Line 46: P has executed LL($newcell \rightarrow Ready$) in Line 43 at time t. By Line 44, *newcell* is valid at t. By Line 45, the LL operation in Line 43 returns *false*. By Lemma 48, if *newcell* is invalid when P executes Line 46, the SC operation returns *false*.

□

Lemma 64 *$\forall$ process P, $\forall c, t$ such that c is reachable and invalid at t: While executing* `promote`*, P does not successfully write to any field in c at t.*

Proof

Line 10: We note that *newcell* is root reachable. P has executed LL($newcell \rightarrow State$) in Line 7 at time t. By Line 8, *newcell* is valid at t. By Line 9, the LL operation in Line 7 returns $\perp$. By Lemma 49, if *newcell* is invalid when P executes Line 10, then the SC operation returns *false*.

Line 11: P writes to $Head[lst]$, which is not a cell.

Line 12: P writes to $head \rightarrow Retired$. By Lemma 44(a), $head \rightarrow Retired = false$ when P begins Line 12. Thus, *head* is valid when P begins Line 12.

Lines 14,16: Lemma 63 applies.

Line 15: Since `promote`(lst) recurses to `promote`(1), Lines 1-13 serve as the basis of an inductive proof.

□

Lemma 65 *$\forall$ process P, $\forall c, t$ such that c is reachable and invalid at t: While executing* `percolateState`*, P does not successfully write to any field in c at t.*

Proof This is an immediate consequence of Lemma 41. □

Lemma 66 *$\forall$ process P, $\forall c, t$ such that c is reachable and invalid at t: While executing* `release`*, P does not successfully write to any field in c at t.*

Proof By Lemma 39(a), we need to consider only Lines 6 and 10.

Line 6: P executes SC($c \rightarrow Free, true$) in Line 6 at time t. Suppose c is invalid at t. Thus, $c \rightarrow Free = true$ at t. By Lemma 39(b), the SC operation in Line 6 returns *false*.

Line 10: analogous to Line 6.

□

Lemma 67 *$\forall$ process P, $\forall c, t$ such that c is reachable and invalid at t: While executing* `apply`*, P does not successfully write to any field in c at t.*

Proof

Line 1: Lemma 60 applies.
Line 2-3: Lemma 64 applies.
Line 4: Lemma 65 applies.
Line 5: Lemma 66 applies.

□

Lemma 68 *No process writes successfully to any invalid, reachable cell.*

Proof This Lemma is a concise re-statement of Lemma 67. □

Lemma 69 *No process writes successfully to any invalid anchor.*

Proof No process accesses an anchor from `percolateState` or `release`, or from Line 6 of `apply`. The argument used to prove Lemma 68 proves this Lemma. □

Lemma 70 *($anchor_i$ is no longer head)$\Rightarrow$ ($anchor_i$ is valid eventually).*

Proof We note that $anchor_i \rightarrow Free = true$ during initialization, and at all times subsequently. The same argument that proves Lemma 44(b) proves that $anchor_i \rightarrow Retired = true$ eventually. Thus, $anchor_i$ is invalid eventually. □

4.5 Bounded Space Complexity (BSC) Implementation

The Bounded Space Complexity (BSC) implementation is the BSC* construction as presented in Figures 4.1 to 4.5, with the modification given in Figures 4.6 and 4.7. We note, in particular, the following change:

> in `announce` and `combine`, replace "allocate a new cell c" with "allocate an unused or invalid cell c :=`selectCell()`".

Thus, in contrast to BSC*, invalid cells are returned by the `selectCell` procedure to either `announce` or `combine` in BSC. Invalid cells are therefore recycled in BSC.

```
initialization
1.   constant δ := 3n + log n
2.   ∀ process P:
3.      A := P's pool of 3δ − 2 cells
           (2 cells of P's are used as anchors)
4.      Z := 2 anchors from P's pool of cells
5.      B, V, I := ∅
end initialization

selectCell () returns *CELL
1.   if |A| = 0
2.      Z := V ∪ B
3.      A := I
4.   if |A| > δ
5.      choose d ∈ A
6.      A := A − {d}, Z := Z ∪ {d}
7.      return d
8.   else
9.      for i := 1, 2, 3:
10.        if |Z| > 0
11.           choose c ∈ Z
12.           Z := Z − {c}
13.           if (c → Free) ∧ (c → Retired)
14.              I := I ∪ {c}
15.           else V := V ∪ {c}
16.     choose d ∈ A
17.     A := A − {d}, B := B ∪ {d}
18.     return d
end selectCell

replace  (c : *CELL)
1.   A := A ∪ {c}
2.   if |A| > δ
3.      Z := Z − {c}
4,   else B := B − {c}
end replace
```

Figure 4.6: Selecting a valid cell

`announce` and `combine`
1(new version). allocate an unused or invalid cell c := `selectCell` () ...

`append` (lst : INTEGER)
11(new version). $c \rightarrow Free := true$, $c \rightarrow Retired := true$, `replace` ($c$)

Figure 4.7: Modifying BSC* to get BSC implementation

4.5.1 `selectCell` and `replace` procedures

We now describe the `selectCell` and `replace` procedures. We note that these are local procedures, *i.e.* they access no shared register. These procedures manage the bounded private pool of cells so that a usable cell is always available to be returned by `selectCell` to either `announce` or `combine`.

Suppose that at any time at most δ cells are valid. (The existence of such a δ will be proved later.) We provide each process with a private pool of 3δ cells, initially divided into two sets: A is the set of usable cells that are ready to be returned by `selectCell`. `selectCell` always returns a cell from A. Z is the set of used cells. A cell in Z may be valid or invalid.

During initialization, A holds all 3δ new cells, and Z is the empty set. As long as $|A| > \delta$, each cell $c \in A$ that `selectCell` returns is moved from A to Z (Lines 4 to 7).

At the point when $|A| = \delta$ (and $|Z| = 2\delta$), each cell $c \in A$ that `selectCell` returns is moved from A to B (Lines 16 to 18). At the same time, with each execution of `selectCell`, three cells in Z are examined as to whether they are valid or invalid (Lines 9 to 15). Valid cells are moved from Z to V; invalid cells are moved from Z to I.

Thus, when $|A|$ reaches zero, all 2δ cells in Z have been examined, and moved to either V or I. Since there are at most δ valid cells at any time, there are at least δ invalid cells in I, ready to be recycled.

At this point, I (the set of invalid cells) becomes the new A (Line 3), and $V \cup B$ becomes the new Z (Line 2). The cycles repeats at this point: `selectCell` returns cells from the new A, which holds at least δ cells that are ready to be recycled.

`replace` is called by Line 11 of `append`, when a cell that was previously returned by `selectCell` fails to become reachable. In this case, the cell is returned to A, from either Z or B, depending on the action taken by the previous `selectCell`.

4.6 Proof of Correctness of BSC

4.6.1 Provisional Correctness Proof

In this section, we give a provisional correctness proof: Suppose that `selectCell` indeed always returns an unused or invalid cell, then the BSC implementation is linearizable and wait-free. Proving this requires the properties of BSC*, proved in the previous sections, concerning reading from, and writing to, invalid cells. Furthermore, an important property (Lemma 71) needs to be proved, before we can assert the provisional correctness of BSC.

Notation 5 *Let $c^{(i)}$ denote the ith incarnation of the cell c.*

In the following, $lcnewcell = c^{(i)}$ (for example) means that the value in *lcnewcell* is c, and that the incarnation of cell c that caused *lcnewcell* to hold the value c is the ith incarnation.

Lemma 71 *Consider the BSC construction.*

- *Suppose that when process P executes Line 26 of* `append`*, $lcnewcell = c^{(i)}, lchild = d^{(j)}$. Then,*

$$(c = d) \Rightarrow (i = j)$$

- *Suppose that when process P executes Line 40 of* `append`*, $rcnewcell = c^{(i)}, rchild = d^{(j)}$. Then,*

$$(c = d) \Rightarrow (i = j)$$

Proof
(a) Let t be the time process P executes Line 23. Since P executes Line 26, both VL operations in Lines 24 and 25 return *true*. By Line 24, $lcnewcell = c^{(i)}$ is valid at t (by Lemma 45). By Line 25, $lchild = d^{(j)}$ is valid at t (by Lemma 45).

Suppose $c = d$. Since the incarnation $c^{(i)}$ pointed to by *lcnewcell* is valid at a time when the incarnation $c^{(j)}$ pointed to by *lchild* is also valid, we have $i = j$.
(b) analogous to part (a).

□

Theorem 5 *Assuming the correctness of* `selectCell`*, the BSC construction is linearizable and wait-free.*

Proof By Lemmas 58 and 59, once a cell (either a reachable cell or an anchor) c is invalid, the contents of the fields in c are no longer needed for our BSC* construction to function correctly. Suppose c is recycled, as specified in the BSC construction. By Lemmas 68 and 69, the content's in c in any of its new incarnations are not corrupted by processes that would have accessed c in its old, invalid incarnation in the BSC* construction.

Finally, pointers to distinct cells in the BSC* construction may point to different incarnations of the same cell in the BSC construction. There are only two places where this feature may cause problem: Lines 26 and 40 of `append`. By Lemma 71, if $lcnewcell = lchild$ in Line 26 (likewise with

the case of $rcnewcell = rchild$ in Line 40), then *lcnewcell* and *lchild* (likewise with *rcnewcell* and *rchild*) point to not only the same cell, but the same incarnation of the same cell.

Thus, even though Line 26 (likewise with Line 40) only checks whether $lcnewcell = lchild$, $lcnewcell = lchild$ in Line 26 implies that both *lcnewcell* and *lchild* point to the same incarnation of the same cell. The check in Line 26 in the BSC construction is therefore identical in function to the check in Line 26 in the BSC* construction. With this fact, we conclude that the BSC construction preserves the correctness of the BSC* construction. The BSC construction is therefore linearizable and wait-free (by Theorem 3). □

4.6.2 Bounding the Number of Valid Cells

Given a process P, a bound exists on the number of cells that are from P's pool, and valid (thus, not available for recycling) at any instant in time. This is obviously an important bound for the purpose of recycling cells. In this section, we prove this bound.

Lemma 72 *Let c be a reachable cell or an anchor. Either c is head forever, or c is invalid eventually.*

Proof
Case 1 c is an anchor.

Lemma 70 applies.

Case 2 c is reachable.

Suppose c is not forever head. By Lemma 39(c), $c \rightarrow Free = true$ eventually. By Lemma 44(b), $c \rightarrow Retired = true$ eventually. Hence, c is invalid eventually.

□

Lemma 73 *At any time t, there are at most $(3n + \log n)$ reachable cells and anchors that are from process P's pool of cells, and valid.*

Proof Let c be a reachable cell or an anchor from P's pool of cells.

Counting P's valid cells that are head

We note that $pos(c) = \lfloor (n + P)/2^i \rfloor$, where $0 \leq i \leq \log n$. Thus, at most $\log n + 1$ cells may be head (and therefore valid) at t, and from P's pool.

Counting P's valid cells that are not head

By Lemma 72, for any cell c that is from P's pool, valid at t, but not head at t, there is some process Q that is executing `apply`, such that Q will be setting either $c \rightarrow Retired$ or $c \rightarrow Free$ to *true* during the `apply`.

We examine the places where $c \rightarrow Retired := true$ and $c \rightarrow Free := true$ occur.

(a) Line 11 of `append`: c is neither reachable, nor an anchor in this case.

(b) Line 12 of `promote`, Lines 28 and 42 of `append`: Each process may have at most one pending operation (in any one of these Lines). Thus, each process can be responsible for at most

one cell that is from P's pool, valid at t, but not head at t. This accounts for a total of at most n cells.

(c) In `release`:

Each process Q can hold up at most $\log n + 1$ cells (in positions $\lfloor (n+Q)/2^i \rfloor$, where $0 \le i \le \log n$). Each of these cells is waiting for Q to complete `release` to become invalid.

However, we are focusing on P's cells, *i.e.* cells that occupy positions $\lfloor (n+P)/2^i \rfloor$, where $0 \le i \le \log n$. If $\lfloor (n+P)/2^i \rfloor$, where $0 \le i \le \log n$, has a leaf descendant position $n+Q$, then Q can hold up one cell of P's in position $\lfloor (n+P)/2^i \rfloor$. For each position $\lfloor (n+P)/2^i \rfloor$, there are $n/(2^{\log n - i})$ leaf descendant positions. Each such leaf descendant position $n+Q$, corresponds to one process Q that may hold up one cell of P's in position $\lfloor (n+P)/2^i \rfloor$.

The total number of P's cells that can be held up in position $\lfloor (n+P)/2^i \rfloor$ because some process has not completed `release` (has not set $Free$ field to $true$) is

$$\begin{aligned} &\le \text{ the number of leaf descendant positions of } \lfloor (n+P)/2^i \rfloor \\ &= n/(2^{\log n - i}) \end{aligned}$$

The total number of P's cells that can be held up because some process has not completed `release` (has not set $Free$ field to $true$) is

$$\begin{aligned} &\le \text{ sum of the numbers of leaf descendant positions of } \lfloor (n+P)/2^i \rfloor, 0 \le i \le \log n \\ &= n(1 + 2^{-1} + 2^{-2} + \cdots 2^{-\log n}) \\ &= 2n - 1 \end{aligned}$$

The total number of P's cells that are valid at t, but not head at t, is therefore at most the sum of the numbers from (b), and (c), $n + 2n - 1 = 3n - 1$.

Since the number of P's cells that are head (hence valid) at t is at most $\log n + 1$, the total number of reachable cells and anchors that are from P's pool and valid at t is at most $3n + \log n$.

□

4.6.3 Properties of `selectCell`

In this section, we prove that the `selectCell` procedure indeed always returns a cell that is suitable for use, *i.e.* it returns either an unused cell or an invalid cell.

Lemma 74 *Suppose at time t:*

- *P executes Line 4 of* `selectCell`*,*
- *$|A| \ge \delta$,*
- *$|Z| + |A| = 3\delta$.*

Let $t_1 > t$ be the earliest time after t when:

- *P executes Line 1 of* `selectCell`*,*
- $|A| = 0$

*Then, when P next executes Line 4 (*i.e. *when P executes Line 4 for the first time after t_1), $|A| \geq \delta, |Z| + |A| = 3\delta$.*

Proof Let t_2, where $t_2 \geq t$, be the earliest time $\geq t$ when (a) P executes Line 4, and (b) $|A| = \delta$. Since during the time interval $[t, t_2]$, every cell that is removed from A is added to Z, $|Z|+|A| = 3\delta$ at t_2.

During the interval $[t_2, t_1]$, all δ cells in A at t_2 become reachable, and each such reachable cell is moved from A to B. Thus, $|B| = \delta$ at t_1.

Since $|Z| + |A| = 3\delta$ at t_2, we have $|Z| = 2\delta$ at t_2. There are at least 3δ executions of the for loop in Lines 9-15 in $[t_2, t_1]$. No cells are added to Z in $[t_2, t_1]$. Thus, every cell c in Z has been examined in $[t_2, t_1]$. If c is valid, c is moved from Z to V. If c is invalid, c is moved from Z to I. Thus, $|V| + |I| = 2\delta$ at t_1. Hence, $|V| + |I| + |B| = 3\delta$ at t_1.

By Lemma 73, there are at most δ valid cells (hence, at least δ invalid cells) in Z at t_2. By Lemma 32(c), a cell that is invalid at t_2 remains invalid throughout $[t_2, t_1]$. Thus, $|I| \geq \delta$ at t_1.

When P completes Line 3 for the first time after t_1, $Z = V \cup B, A = I$. Hence, $|A| \geq \delta, |Z| + |A| = |V| + |B| + |I| = 3\delta$.

□

Lemma 75 *If $|A| = 0$ when P executes Line 1 of* `selectCell`*, then when P next executes Line 4, $|A| \geq \delta, |Z| + |A| = 3\delta$.*

Proof This Lemma is proved by induction, using Lemma 74.
Induction Basis Consider P's first invocation of `selectCell`.

$|A| = 3\delta - 2 \geq \delta$, $|Z| = 2$ (A, Z are as assigned during initialization), when P executes Line 4. Therefore Lemma 75 holds for the first time when (a)P executes Line 1, (b)$|A| = 0$ (by Lemma 74).
Induction Step Suppose that Lemma 75 holds for the ith time when (a)P executes Line 1, (b)$|A| = 0$. Then, by Lemma 74, Lemma 75 holds for the $(i+1)$th time when (a)P executes Line 1, (b)$|A| = 0$.

□

Lemma 76 `selectCell` *always returns either an unused cell or an invalid reachable cell.*

Proof Suppose that `selectCell` returns d. Then, d was in A. Consider the assignment of A during the initialization, and in Line 3 of `selectCell`. Since I contains only invalid cells, A contains only invalid cells after Line 3 of `selectCell` is executed. Thus, d is either an unused cell or an invalid reachable cell.

By Lemma 75, when P executes Lines 5, 16 of `selectCell` to choose $d \in A$, $A \neq \emptyset$. Thus, `selectCell` always returns a cell. Hence, our Lemma holds. □

4.6.4 Correctness of the BSC Implementation

Given the provisional correctness, and the correctness of `selectCell`, we reach an immediate conclusion that BSC is linearizable and correct.

However, we need to further specify the details of implementing the data structures in `selectCell`, in order to conclude that the local time complexity of `selectCell` is $O(1)$.

We note that the operations performed on P's local sets S, where S may be A, B, Z, V, or I, are (a)choose an element c from set S, and move c from S to S', (b)$Z := V \cup B$ (Line 2), (c) $A := I$ (Line 3).

To achieve $O(1)$ local time complexity for each execution of `selectCell` and `replace`, we use the following implementation:

A set S, where S may be B, V, or I, is implemented as an array of size 3δ, with a variable $size(S)$ that indicates the index of the last element in S. We always remove or add the last element of the array.

There are six arrays: $B_0, B_1, V_0, V_1, I_0, I_1$, and two flags: $ZFlag$, and $AFlag$. A, B, Z, V, and I are implemented as follows:

When $ZFlag = i (i = 0, 1)$, Z is the combination of arrays V_i and B_i (in that order); V is the array V_{1-i}; B is the array B_{1-i}. When $AFlag = i(1 = 0, 1)$, A is the array I_i; I is the array I_{1-i}.

In Line 2, $Z := V \cup B$ is implemented by $ZFlag := 1 - ZFlag$. In Line 3, $A := I$ is implemented by $AFlag := 1 - AFlag$.

With this implementation, we have:

Lemma 77 *The local time complexity of a call to either* `selectCell` *or* `replace` *is* $O(1)$.

Theorem 6 *(a)The BSC construction is linearizable and wait-free.*
(b)The shared-access time complexity and the local time complexity of the BSC construction are both $O(\log^2 n)$.
(c)The BSC construction requires $3n(3n + \log n)$ *shared cells and* $2n - 1$ *shared pointers to cells.*

Proof
(a) By Lemma 76, `selectCell` is correct. By Theorem 5, the BSC construction is linearizable and wait-free.
(b) By Theorem 4 and Lemma 77.
(c) Each process has a pool of $3(3n + \log n)$ cells. Thus, there are $3n(3n + \log n)$ shared cells in total. In addition, there are $2n - 1$ shared pointers to cells: $Head[lst], 1 \leq lst \leq 2n - 1$.

□

Chapter 5

Contention-Sensitive Implementation

5.1 General Principles

Our objective is to enhance the BSC implementation, so that when there are few concurrent invocations, a process can complete an invocation of `apply` quickly. Our general approach is as follows: We introduce $\log^2 n$ new node positions to the binary tree defined in Chapter 3. These are the *middle children* of positions 1,2, $\cdots, k, \cdots, \log^2 n$. They are numbered $2.5, 4.5, \cdots, 2k + .5, \cdots, 2 * \log^2 n + .5$ respectively. We note that these new positions are closer than the leaf positions $n, n+1, \cdots, 2n-1$ are to the root.

Concurrent processes compete for the right to install their operation cells at one of these new node positions, and then traverse up from that position to the root position. A process that has its operation cell installed as a middle child is able to avoid traversing the full length of the path leading from a leaf to the root.

Let OP be an invocation of `apply`. Recall that we defined the *contention experienced by* OP, denoted by n_c, as the maximum number of concurrent invocations during the interval of OP's execution. When P is the only process executing an invocation of `apply`, P installs its operation cell at the middle child position of the root, thus fully exploiting the benefit of no contention.

In order to gain the right to install its operation cell at a middle child position (which is close to the root), process P executes the `getToken` procedure. [1] Each of the middle child positions $2k+.5$ has a corresponding token k. If P gets token k, then P installs its operation cell at position $2k+.5$. If P fails to get any of the $\log^2 n$ tokens, then P proceeds to execute its `apply` as specified by BSC. In other words, P installs its operation cell at position $n + P$, and traverse up to the root position.

We will show that if P gets token m, then the contention experienced by P's invocation is at least m, *i.e.* $n_c \geq m$. Furthermore, if P gets token m, the time complexity of P's invocation of `apply` is dominated by the time taken to execute `getToken` (*i.e.* the time taken to get the token), which equals $O(m)$. Since $m \leq n_c, m \leq \log^2 n$, we achieve a time complexity of $O(\min(n_c, \log^2 n))$.

[1] The idea of processes competing for tokens that signify possession of a node close to the root was first introduced by Afek, Dauber, and Touitou [ADT95].

```
type INTEGER+ := {1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, ···}

initialization
∀i, 1 ≤ i ≤ 2n-1 : anchor_i : *CELL
∀j, 1 ≤ j ≤ log² n : anchor_{2j+.5} : *CELL

1.    for i := 1 to 2n-1:
2.       anchor_i → Next := ⊥, anchor_i → Free := true,
         anchor_i → Retired := false, Head[i] := anchor_i
3.    for j := 1 to log² n:
4.       i := 2 * j + .5
5.       anchor_i → Next := ⊥, anchor_i → Free := true,
         anchor_i → Retired := false, Head[i] := anchor_i
6.    anchor_1 → State := INITIALSTATE
7.    anchor_1 → Op := ⊥
   end initialization


   apply (P : INTEGER, op : OP, 𝒪) returns RES
1.    lst := getToken(P)
2.    opcell := announce (op, lst)
3.    for i := 0 to ⌊log lst⌋
4.       promote (⌊lst/2^i⌋)
5.    percolateState (opcell, lst)
6.    response:= δ_resp (opcell → State, op)
7.    release (opcell, lst, middle)
8.    if lst ≠ n+P
9.       token_⌊lst/2⌋ := 0
10.   return response
   end apply
```

Figure 5.1: Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9)

```
   promote (lst : INTEGER+)
1.    if lst = 1
2.       head := LL(Head[lst])
3.       newcell := head → Next
4.       if newcell = ⊥ return
5.       if newcell → Ready = false return
6.       newstate := δ_state (head → State, head → Op)
7.       newcellstate := LL(newcell → State)
8.       if not VL(Head[lst]) return
9.       if newcellstate = ⊥
10.         SC(newcell → State, newstate)
11.      if SC(Head[lst], newcell)
12.         head → Retired := true
13.      return
14.   append (⌊lst/2⌋)
15.   promote (⌊lst/2⌋)
16.   append (⌊lst/2⌋)
   end promote
```

Figure 5.2: Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9)

```
    append (lst : INTEGER+)
1.    head := LL(Head[lst])
2.    newcell := LL(head → Next)
3.    if not VL(Head[lst]) return
4.    if newcell = ⊥
5.       (lc, lop) := readyOrphan (2 ∗ lst)
6.       (mc, mop) := readyOrphan (2 ∗ lst + .5)
7.       (rc, rop) := readyOrphan (2 ∗ lst + 1)
8.       if (lc ≠ ⊥) or (mc ≠ ⊥) or (rc ≠ ⊥)
9.          c := combine (lc, lop, mc, mop, rc, rop)
10.         result := SC (head → Next, c)
11.         if not result
12.            c → Free := true, c → Retired := true, replace (c)
13.      newcell := head → Next
14.      if not VL(Head[lst]) return
15.      if newcell = ⊥ return

16.   lchild := newcell → LC
17.   if not VL(Head[lst]) return
18.   if lchild ≠ ⊥
19.      lcparent := LL(lchild → Parent)
20.      if not VL(Head[lst]) return
21.      if lcparent = ⊥
22.         SC(lchild → Parent, newcell)
23.      lchead := LL(Head[2*lst])
24.      lcnewcell := lchead → Next
25.      if not VL(Head[2*lst]) go to K
26.      if not VL(Head[lst]) return
27.      if lcnewcell = lchild
28.         if SC(Head[2*lst], lchild)
29.            lchead → Retired := true
    (continue on next Figure)
```

Figure 5.3: Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9)

```
   append (lst : INTEGER+)
       (continued from previous Figure)

30.K  mchild := newcell → MC
31.   if not VL(Head[lst]) return
32.   if mchild ≠ ⊥
33.      mcparent := LL(mchild → Parent)
34.      if not VL(Head[lst]) return
35.      if mcparent = ⊥
36.         SC(mchild → Parent, newcell)
37.      mchead := LL(Head[2*lst+ .5])
38.      mcnewcell := mchead → Next
39.      if not VL(Head[2*lst+ .5]) go to L
40.      if not VL(Head[lst]) return
41.      if mcnewcell = mchild
42.         if SC(Head[2*lst+ .5], mchild)
43.            mchead → Retired := true

44.L  rchild := newcell → RC
45.   if not VL(Head[lst]) return
46.   if rchild ≠ ⊥
47.      rcparent := LL(rchild → Parent)
48.      if not VL(Head[lst]) return
49.      if rcparent = ⊥
50.         SC(rchild → Parent, newcell)
51.      rchead := LL(Head[2*lst+1])
52.      rcnewcell := rchead → Next
53.      if not VL(Head[2*lst+1]) go to M
54.      if not VL(Head[lst]) return
55.      if rcnewcell = rchild
56.         if SC(Head[2*lst+1], rchild)
57.            rchead → Retired := true

58.M  ready := LL(newcell → Ready)
59.   if not VL(Head[lst]) return
60.   if ready = false
61.      SC(newcell → Ready, true)
   end append
```

Figure 5.4: Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9)

```
   readyOrphan (lst : INTEGER⁺) returns (*CELL, OP)
1.   if (lst= 2 * ⌊lst/2⌋ + .5) and (lst > 1 + 2 * log² n) return (⊥, ⊥)
2.   head := LL(Head[lst])
3.   newcell := head → Next
4.   if newcell = ⊥  return (⊥, ⊥)
5.   if newcell → Ready = false return (⊥, ⊥)
6.   newop := newcell → Op
7.   if not VL(Head[lst]) return (⊥, ⊥)
8.   return (newcell, newop)
   end readyOrphan

   announce (op : OP, lst : INTEGER⁺) returns *CELL
1.   allocate an unused or invalid cell c := selectCell () and initialize it as follows:
       c → Parent := ⊥, c → Next := ⊥, c → State := ⊥
       c → LC := ⊥, c → MC := ⊥, c → RC := ⊥,
       c → Op := op, c → Lop := ⊥, c → Mop := ⊥, c → Ready := true
       c → LDone := true, c → MDone := false, c → RDone := true,
       c → Free := false, c → Retired := false
2.   head := Head[lst]
3.   if LL(head → Next) = ⊥
4.     SC(head → Next, c)
5.   return c
   end announce
```

Figure 5.5: Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9)

```
combine (lc : *CELL, lop : OP, mc : *CELL, mop : OP, rc : *CELL, rop : OP) returns *CELL
1.  allocate an unused or invalid cell c := selectCell () and initialize it as follows:
      c → Parent := ⊥, c → Next := ⊥, c → State := ⊥
      c → LC := lc, c → MC := mc, c → RC := rc,
      c → Lop := lop,
      c → Mop := lop ⊗ mop
      c → Op := (c → Mop) ⊗ rop
      c → Ready := false, c → Free := false, c → Retired := false
      if lc ≠ ⊥
        c → LDone := false
      else c → LDone:= true
      if mc ≠ ⊥
        c → MDone := false
      else c → MDone := true
      if rc ≠ ⊥
        c → RDone := false
      else c → RDone := true
2.  return c
end combine

percolateState (c : *CELL, lst : INTEGER+)
1.  if lst = 1 return
2.  p := c → Parent
3.  percolateState (p, ⌊lst/2⌋)
4.  if lst = 2 ∗ ⌊lst/2⌋
5.      c → State := p → State, return
6.  if lst = 2 ∗ ⌊lst/2⌋ + .5
7.      c → State := δ_state (p → State, p → Lop), return
8.  c → State := δ_state (p → State, p → Mop), return
end percolateState
```

Figure 5.6: Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9)

```
   release (c : *CELL, lst : INTEGER+, dir : {left, middle, right})
1.   p := c → Parent
2.   LL(c → Free)
3.   if dir = left
4.      c → LDone := true
5.      if c → MDone and c → RDone
6.         if SC(c → Free, true)
7.            ascend (p, lst)
8.      return
9.   if dir = middle
10.     c → MDone := true
11.     if c → LDone and c → RDone
12.        if SC(c → Free, true)
13.           ascend (p, lst)
14.     return
15.  c → RDone := true
16.  if c → LDone and c → MDone
17.     if SC(c → Free, true)
18.        ascend (p, lst)
19.  return
   end release
```

```
   ascend (c : *CELL, lst : INTEGER+)
1.   if lst =1 return
2.   if lst = 2 ∗ ⌊lst/2⌋
3.      release (c, ⌊lst/2⌋, left), return
4.   if lst = 2 ∗ ⌊lst/2⌋ + .5
5.      release (c, ⌊lst/2⌋, middle), return
6.   release (c, ⌊lst/2⌋, right), return
   end ascend
```

Figure 5.7: Contention-sensitive construction for closed object $\mathcal{O}$ (Figures 5.1 to 5.9)

5.2 The Implementation

We highlight some noteworthy aspects of our contention-sensitive implementation, which we present in Figures 5.1 to 5.9.

(1). In addition to the possible node positions as specified by a binary tree of height $\log n$ (see Figure 3.1(a)), we have $\log^2 n$ new node positions: $2j + .5$, for $j = 1, 2, \cdots, \log^2 n$. A node at position $2j + .5$ is the *middle child* of the node at position j.

(2). In the new construction, each cell in List[lst], $1 \leq lst \leq \log^2 n$, has three children: a left child and a right child as in the original construction, and a middle child, which is either $\perp$ or a cell in list $2 * lst + .5$. The middle child, if present, is a leaf.

(3). In the BSC construction, when a process P invokes `apply`$(P, op, \mathcal{O})$, it stores op in a cell c, installs c (as a leaf) in the List[$n + P$] (which belongs exclusively to P), and works towards making c a part of a fully formed tree (*i.e.*, until c has $\log n$ ancestors).

In the new construction, when P invokes `apply`$(P, op, \mathcal{O})$, it first executes the `getToken` procedure (Figure 5.8). Suppose P succeeds in getting a token m. It stores op in a cell c, and installs c (as a leaf) in List[$2m + .5$]. P then works towards getting c a parent in List[m], a grandparent in List[$\lfloor m/2 \rfloor$] and so on, until c has an ancestor in List[1]. Then, P computes the response to its operation, and releases token m. P's `getToken` takes $O(m)$ time and the rest of the work takes $O(\log^2 m)$ time. We will show that $m \leq n_c$ (n_c is the contention experienced by P's invocation). Further, $m \leq \log^2 n$. Thus, the time complexity of our construction is $O(\min(n_c, \log^2 n))$.

If P does not succeed in getting a token, then $n_c \geq \log^2 n + 1$. In this case, P performs its operation as in the BSC construction, by installing an operation cell in List[$n + P$] and working towards the root from there. In this case, the total time spent is $O(\log^2 n)$. Since $n_c \geq \log^2 n + 1$, we have the desired $O(\min(n_c, \log^2 n))$ time complexity bound.

(4). We enhance the `append` procedure so that, when attempting to append a new cell c to List[lst], $1 \leq lst \leq \log^2 n$, in addition to parenting any ready orphans in List[$2 * lst$] and List[$2 * lst + 1$], c will also adopt as its middle child any ready orphan in List[$2 * lst + .5$].

(5). Each cell c has three fields that record operations: The *Lop* field records the operation of c's left child. The Mop field records the operation that results from combining the operations of c's left and middle children. The *Op* field records the operation that results from combining the operations of c's left, middle, and right children. *Lop* and Mop fields are useful in the `percolateState` procedure, when computing the *State* fields of c's children, given the *State* field of c. Let s be the value in the *State* field of c. Then, the *State* field of c's left child is s. The *State* field of c's middle child is the state that results from applying c's *Lop* to the implemented object in state s. The *State* field of c's right child is the state that results from applying c's Mop to the implemented object in state s.

5.3 Proof of Correctness

With the exception of `getToken`, presented in Figure 5.8. the proof of linearizability and wait-freedom of BSC in Chapter 4 carries over to our construction, with obvious minor changes. We therefore prove only the relevant properties of `getToken` here.

```
   getToken(P : PROCESSES) returns INTEGER+
1.   for i := 1 to log^2 n
2.     if LL(token_i) = 0
3.       if SC(token_i, 1)
4.         return 2 * i + .5
5.   return n+P
   end getToken
```

Figure 5.8: Contention-sensitive construction for closed objects: `getToken` (Figures 5.1 to 5.9)

5.3.1 Proof of Token Scheme

In `getToken`, process P tries to win a token $token_i$ by executing the **for** loop in Lines 1 to 4 with index i. P iterates the **for** loop with index $i = 1, 2, \cdots, \log^2 n$. In each iteration, P executes LL($token_i$) in Line 2. If $token_i = 0$ in Line 2, then no process has yet won $token_i$. In this case, P executes SC($token_i, 1$) in Line 3. If the SC operation returns *true*, then P has won $token_i$. Clearly, at most one process can win any $token_i$. If P wins $token_i$, then P's `getToken` returns $2i+.5$, the middle child position that P is now entitled to install its *opcell*, by virtue of P's having won $token_i$. If P did not win any of $token_i, 1 \leq i \leq \log^2 n$, then P's `getToken` returns $n + P$, which is the leaf position that, in all preceding algorithms, P installs its *opcell*.

Let $token_i$ be the token that P has won in `getToken`. Then, P releases $token_i$ in Line 9 of `apply`, by setting $token_i$ to 0, just before `apply` terminates.

We say that P *misses* $token_i$ if when P executes the **for** loop with $i = k$, P either finds $token_k = 1$ in Line 2, or its SC operation in Line 3 returns *false*. Formally,

Definition 19

- *We say that P* misses $token_k$ *if and only if* $\forall k, 1 \leq k \leq \log^2 n - 1$*: if process P begins Line 1 of* `getToken` *with index* $i = k + 1$.
- *If process P executes Line 5 of* `getToken`*, then we say that P* misses $token_{\log^2 n}$.

Notation 6

- *Let t_k^P be the time P begins Line 1 of* `getToken` *with index $i = k$ (if P ever does so).*
- *Let $t_{1+\log^2 n}^P$ be the time P begins Line 5 of* `getToken` *with index $i = k$ (if P ever does so).*
- *Let process P miss $token_k$, where $1 \leq k \leq \log^2 n$. Then, t_{k+1}^P is well-defined. Let I_k^P denote the time interval $[t_1^P, t_{k+1}^P]$.*

```
    initialization
1.    constant δ := 8n log log n
2.    ∀ process P:
3.        A := P's pool of 3δ − 3 cells
            (3 cells of P's are used as anchors)
4.        Z := 3 anchors from P's pool of cells
5.        B, V, I := ∅
    end initialization

    selectCell () returns *CELL
1.    if |A| = 0
2.        Z := V ∪ B
3.        A := I
4.    if |A| > δ
5.        choose d ∈ A
6.        A := A − {d}, Z := Z ∪ {d}
7.        return d
8.    else
9.        for i := 1, 2, 3:
10.           if |Z| > 0
11.               choose c ∈ Z
12.               Z := Z − {c}
13.               if (c → Free) ∧ (c → Retired)
14.                   I := I ∪ {c}
15.               else V := V ∪ {c}
16.       choose d ∈ A
17.       A := A − {d}, B := B ∪ {d}
18.       return d
    end selectCell

    replace  (c : *CELL)
1.    A := A ∪ {c}
2.    if |A| > δ
3.        Z := Z − {c}
4,    else B := B − {c}
    end replace
```

Figure 5.9: Contention-sensitive construction for closed objects: (Figures 5.1 to 5.9)

Informally, if P misses $token_k$, then $[t_k^P, t_{k+1}^P]$ is an interval within which P executed the **for** loop with $i = k$. In other words, P tries and fails to gain $token_k$ during $[t_k^P, t_{k+1}^P]$. During $I_k^P = [t_1^P, t_{k+1}^P]$, P tries and fails to gain $token_1, token_2, \cdots, token_k$.

Definition 20 *$\forall k, 1 \leq k \leq \log^2 n$, if process P executes a successful SC($token_k, 1$) in Line 3 of* `getToken`*, then we say that P* wins *$token_k$.*

Our objective is to prove Lemma 79, which asserts that if P wins $token_{k+1}$, then there must be at least k invocations that are concurrent with P's invocation. Lemma 78 is a technical Lemma that leads easily to Lemma 79.

Lemma 78 asserts the following: Suppose that P misses $token_k$. Then, there is an instant in time τ_k^P during the interval I_k^P (an interval during which P tries and fails to gain $token_1, token_2, \cdots, token_k$) such that at τ_k^P, there are k distinct active processes, each having won one of the following k tokens: $token_1, token_2, \cdots, token_k$.

Lemma 78 *Let process P miss $token_k$, where $1 \leq k \leq \log^2 n$. Then, there exists a time $\tau_k^P \in I_k^P$ such that $\forall j, 1 \leq j \leq k : \exists$ process Q_j such that:*

- *Q_j is active,* i.e. *executing* `apply`*, at τ_k^P,*
- *Q_j wins $token_j$.*

Proof We proceed by induction on k.
Induction Basis $k = 1$.
In this case, P misses $token_1$, We need to prove that $\exists \tau_1^P \in I_1^P = [t_1^P, t_2^P], \exists Q_1$, such that:

- Q_1 is active at τ_1^P,
- Q_1wins $token_1$.

Since P misses $token_1$, there exists a time $\tau_1^P \in I_1^P = [t_1^P, t_2^P]$ when $token_1 = 1$. Let Q_1 be the last process to execute a successful SC($token_1, 1$) in Line 3 of `getToken` before τ_1^P. Thus, Q_1 wins $token_1$, and Q_1 is active at τ_1^P.
Induction Step The Induction Hypothesis is that Lemma 78 holds for $k, 1 \leq k \leq m$. We now prove that Lemma 78 holds for $k = m + 1$.

Let process P miss $token_{m+1}$, where $2 \leq m + 1 \leq \log^2 n$. Since P misses $token_{m+1}$, there exists a time $\tau' \in [t_{m+1}^P, t_{m+2}^P]$ when $token_{m+1} = 1$. Let Q_{m+1} be the last process to execute a successful SC($token_{m+1}, 1$) in Line 3 of `getToken` before τ'. Thus, Q_{m+1} wins $token_{m+1}$, and Q_{m+1} is active at τ'.

Since Q_{m+1} wins $token_{m+1}$, Q_{m+1} misses $token_m$. By the Induction Hypothesis, there exists $\tau_m^{Q_{m+1}} \in I_m^{Q_{m+1}} = [t_1^{Q_{m+1}}, t_{m+1}^{Q_{m+1}}]$, satisfying Lemma 78. Since P misses $token_{m+1}$, P misses $token_m$. By the Induction Hypothesis, there exists $\tau_m^P \in I_m^P = [t_1^P, t_{m+1}^P]$, satisfying Lemma 78. Let $\tau_{m+1}^P = \max(\tau_m^P, \tau_m^{Q_{m+1}})$.

We now proceed to prove that τ_{m+1}^P satisfies Lemma 78.

Claim 1 $\tau_{m+1}^P \in I_{m+1}^P = [t_1^P, t_{m+2}^P]$.

Proof of Claim We have $t_1^P \leq \tau_m^P \leq t_{m+1}^P$, $t_{m+1}^P < t_{m+2}^P$, $\tau_m^{Q_{m+1}} < \tau' < t_{m+2}^P$. Since $t_1^P \leq \tau_m^P < t_{m+2}^P, \tau_m^{Q_{m+1}} < t_{m+2}^P$, we have $t_1^P \leq \max(\tau_m^P, \tau_m^{Q_{m+1}}) < t_{m+2}^P$, *i.e.* $\tau_{m+1}^P \in [t_1^P, t_{m+2}^P]$.

□

We need to define $Q_j, \forall j, 1 \leq j \leq m+1$, for the purpose of Lemma 78. Q_{m+1} has been defined above. If $\tau_{m+1}^P = \tau_m^P$, then we let $Q_1, Q_2, \cdots, Q_m$ be the m processes that are active at τ_m^P, and are guaranteed to exist by the Induction Hypothesis (as stated above) applied to P . If $\tau_{m+1}^P = \tau_m^{Q_{m+1}}$, then we let $Q_1, Q_2, \cdots, Q_m$ be the m processes that are active at $\tau_m^{Q_{m+1}}$, and are guaranteed to exist by the Induction Hypothesis applied to Q_{m+1}.

Claim 2 $\forall j, 1 \leq j \leq m, \exists Q_j$ *such that:*

- Q_j *is active at* τ_{m+1}^P*,*
- Q_j *wins* $token_j$*.*

Proof of Claim This is an immediate consequence of our choice of $Q_1, Q_2, \cdots, Q_m$.

□

Claim 3 Q_{m+1} *is active at* τ_{m+1}^P*.*

Proof of Claim If $\tau_{m+1}^P = \tau_m^{Q_{m+1}}$, then our Claim is obvious. Suppose $\tau_{m+1}^P = \tau_m^P$, *i.e.* $\tau_m^{Q_{m+1}} < \tau_m^P$. We must show that Q_{m+1} is active at τ_m^P. We note that Q_{m+1} is active throughout the interval $[\tau_m^{Q_{m+1}}, \tau']$. Since $\tau_m^{Q_{m+1}} < \tau_m^P < \tau'$, Q_{m+1} is active at τ_m^P

□

By Claims 1, 2, 3, and the fact that Q_{m+1} wins $token_{m+1}$, Lemma 78 holds for $k = m+1$. This complete the proof of Lemma 78.

□

Lemma 79 *Let k be such that $k \geq 0$. Suppose that process P wins $token_{k+1}$. Then, P's invocation is concurrent with at least k invocations.*

Proof Since P wins $token_{k+1}$, P has previously missed $token_k$. By Lemma 78, there exists a time $\tau_k^P \in I_k^P$ such that $\forall j, 1 \leq j \leq k : \exists$ process Q_j such that:

- Q_j is active, *i.e.* executing `apply`, at τ_k^P,
- Q_j wins $token_j$.

We note that, by Notation 6, P is active, *i.e.* executing `apply`, at τ_k^P. Since $\forall j, 1 \leq j \leq k : Q_j$ wins $token_j$, we know that $Q_1, Q_2, \cdots, Q_k$, and P are $k+1$ distinct processes. Furthermore, as we just observed, all $k+1$ processes are active at τ_k^P. Hence, P's invocation is concurrent with at least k invocations, *i.e.* the invocations by $Q_1, Q_2, \cdots, Q_k$, at τ_k^P. This completes our proof.

□

5.3.2 Bounding the Number of Valid Cells

Given a process P, a bound exists on the number of cells that are from P's pool, and valid (thus, not available for recycling) at any instant in time. This is obviously an important bound for the purpose of recycling cells. In this section, we prove this bound.

Lemma 80 *Suppose that $n \geq 16$. At any time t, there are at most $8n \log \log n$ reachable cells and anchors that are from process P's pool of cells, and valid.*

Proof Let c be a reachable cell or an anchor from P's pool of cells.

Counting P's valid cells that are head

Case 1

c is allocated during an invocation of `apply` when P wins a token.

We have $pos(c) = i$, or $pos(c) = 2i + .5$, where $1 \leq i \leq \log^2 n$. Thus, at most $2 \log^2 n$ such cells may be head (and therefore valid) at t, and from P's pool.

Case 2

c is allocated during an invocation of `apply` when P does not win any token.

We have $pos(c) = \lfloor (n + P)/2^i \rfloor$, where $0 \leq i \leq \log n$. Thus, at most $\log n + 1$ such cells may be head (and therefore valid) at t, and from P's pool.

Hence, the total number of P's cells that are head (and therefore valid) at t is $2 \log^2 n + \log n$ (we omit one cell, because of the double counting of the root position). Since we assume that $n \geq 16$, we have $n \geq \log^2 n > \log n$. Hence, $2 \log^2 n + \log n \leq 3n$.

Counting P's valid cells that are not head

By Lemma 72, for any cell c that is from P's pool, valid at t, but not head at t, there is some process Q that is executing `apply`, such that Q will be setting either $c \rightarrow Retired$ or $c \rightarrow Free$ to $true$ during the `apply`.

We examine the places where $c \rightarrow Retired := true$ and $c \rightarrow Free := true$ occur.

(a) Line 12 of `append`: c is neither reachable, nor an anchor in this case.

(b) Line 12 of `promote`, Lines 29 and 43 of `append`: Each process may have at most one pending operation (in any one of these Lines). Thus, each process can be responsible for at most one cell that is from P's pool, valid at t, but not head at t. This accounts for a total of at most n cells.

(c) In `release`:

Each process Q can hold up either (a) at most $\log n + 1$ cells in positions $\lfloor (n + Q)/2^i \rfloor$, where $0 \leq i \leq \log n$, or (b) at most $\log \log^2 n$ cells along the path from a middle child position to the root position. Each of these cells is waiting for Q to complete `release` to become invalid.

However, we are focusing on P's cells c:

Case 1

Consider cells c such that either $pos(c) = i$, or $pos(c) = 2i + .5$, where $1 \le i \le \log^2 n$. Among these cells, only cells c such that $pos(c) = i$, where $1 \le i \le \log^2 n$, can be waiting for Q to complete `release` to become invalid. Each process Q can hold up cells in at most $\log\log^2 n$ of these $\log^2 n$ positions. Thus, the total number of valid cells waiting for Q to complete `release` is $n \log\log^2 n = 2n \log\log n$

Case 2

We consider P's cells c that occupy positions $\lfloor (n + P)/2^i \rfloor$, where $0 \le i \le \log n$, such that $pos(c) > \log^2 n$. In other words, we exclude all positions from 1 to $log^2 n$. Therefore, we are considering cells that occupy positions $\lfloor (n+P)/2^i \rfloor$, where $0 \le i \le w$, $w = \log n - \log\log^2 n - 1$.

If c occupies such a position, then, the argument used in Lemma 73 applies here. If $\lfloor (n+P)/2^i \rfloor$, where $0 \le i \le w$, has a leaf descendant position $n + Q$, then Q can hold up one cell of P's in position $\lfloor (n + P)/2^i \rfloor$. For each position $\lfloor (n + P)/2^i \rfloor$, there are $n/(2^{\log n - i})$ leaf descendant positions. Each such leaf descendant position $n + Q$, corresponds to one process Q that may hold up one cell of P's in position $\lfloor (n + P)/2^i \rfloor$.

The total number of P's cells that can be held up in position $\lfloor (n + P)/2^i \rfloor$ because some process has not completed `release` (has not set $Free$ field to $true$) is

$$
\begin{aligned}
&\le \text{ the number of leaf descendant positions of } \lfloor (n+P)/2^i \rfloor \\
&= n/(2^{\log n - i})
\end{aligned}
$$

The total number of P's cells that can be held up because some process has not completed `release` (has not set $Free$ field to $true$) is

$$
\begin{aligned}
&\le \text{ sum of the numbers of leaf descendant positions of } \lfloor (n+P)/2^i \rfloor, 0 \le i \le w \\
&\le \text{ sum of the numbers of leaf descendant positions of } \lfloor (n+P)/2^i \rfloor, 0 \le i \le \log n \\
&= n(1 + 2^{-1} + 2^{-2} + \cdots 2^{-\log n}) \\
&= 2n - 1
\end{aligned}
$$

The total number of P's cells that are valid at t, but not head at t, is therefore at most the sum of the numbers from (b), and (c), $n + (2n - 1) + 2n \log\log n \le 5n \log\log n$.

Since the number of P's cells that are head (hence valid) at t is at most $3n$, the total number of reachable cells and anchors that are from P's pool and valid at t is at most $3n + 5n \log\log n \le 8n \log\log n$.

□

5.3.3 Proof of Correctness

We are now in a position to prove the following Theorem that summarizes all the important properties of our contention-sensitive construction.

Theorem 7 *(a)The contention-sensitive construction is linearizable and wait-free.*
(b)The shared-access time complexity and the local time complexity of the contention-sensitive construction are both $O(min(n_c, \log^2 n))$.
(c)The contention-sensitive construction requires $24n^2 \log\log n$ shared cells, $2n + \log^2 n$ shared pointers to cells, and $\log^2 n$ shared bits.

Proof
(a) Wait-freedom is easily seen from the algorithm. The proof of Theorem 6(a), which asserts that the BSC construction is linearizable, carries over, with easy modifications, to the contention-sensitive implementation.
(b) Consider process P executing `apply`. Suppose P succeeds in getting $token_m$. It stores *op* in a cell c, and installs c (as a leaf) in List$[2m+.5]$. P then works towards getting c a parent in List$[m]$, a grandparent in List$[\lfloor m/2 \rfloor]$ and so on, until c has an ancestor in List[1]. Then, P computes the response to its operation, and releases $token_m$. P's `getToken` takes $O(m)$ time and the rest of the work takes $O(\log^2 m)$ time. By Lemma 79, P's operation is concurrent with at least $m-1$ invocations. In other words, $m \leq n_c$, where n_c is the contention experienced by P's invocation. Further, $m \leq \log^2 n$. Thus, the time complexity of our construction is $O(\min(n_c, \log^2 n))$.

If P does not succeed in getting a token, then, by Lemma 78, $n_c \geq \log^2 n + 1$. In this case, P installs an operation cell in List$[n+P]$ and working towards the root from there. The total time spent is $O(\log^2 n)$. Since $n_c \geq \log^2 n + 1$, we have the desired $O(\min(n_c, \log^2 n))$ time complexity in this case, too.
(c) By Lemma 80, we see that setting $\delta = 8n \log\log n$ in `selectCell` results in a `selectCell` that always returns a unused or invalid cell. Thus, our contention-sensitive implementation requires a total of $3n(8n \log\log n) = 24n^2 \log\log n$ shared cells.

In addition, the $\log^2 n$ new middle child positions require $\log^2 n$ additional shared pointers to cells. Hence, a total of $2n + \log^2 n$ shared pointers to cells are needed.

Finally, there are $\log^2 n$ tokens, $token_1, token_2, \cdots, token_{\log^2 n}$. Each token is a shared bit. Thus, we have part(c).

□

In conclusion, we have presented, in Figures 5.1 to 5.9, a contention-sensitive, wait-free, linearizable construction for closed objects, with time complexity of $O(\min(n_c, \log^2 n))$, and $O(n^2 \log\log n)$ shared memory requirement.

Part II

Lower Bounds for Nonblocking Implementations

Chapter 6

Time and Space Lower Bounds for Nonblocking Implementations

6.1 The Lower Bound

This section has three parts. Section 6.1.1 illustrates the main ideas of our lower bound technique for the special case of implementing an increment object. Our lower bound applies not just to implementations of the increment object, but to implementations of a large class of objects, which we call perturbable objects. Section 6.1.2 defines the class of perturbable objects. Section 6.1.3 proves the lower bound for any implementation of any perturbable object.

6.1.1 The Intuition

To illustrate the main ideas of the lower bound proof, we provide below a proof sketch for a simple case of the full result. Consider any deterministic implementation of an increment object $\mathcal{O}$, shared by $p_1, \ldots, p_n$, from swap objects. We will prove that the space and the shared-access time complexity of the implementation are at least $n-1$. (The full result is more general in three ways: it applies to randomized implementations; it applies to implementations of any perturbable object, not just the increment object; and it applies even if base objects include resettable consensus objects and any historyless objects.)

Consider Scenario *0* in which p_n initiates a read operation on $\mathcal{O}$ and runs alone. We claim that p_n accesses some base object, before completing this read operation on $\mathcal{O}$. For a proof, suppose the claim is false. Then, p_n cannot distinguish Scenario *0* from Scenario *0'* in which some process completes an increment operation on $\mathcal{O}$ before p_n starts taking steps. Yet, correctness requires p_n's read operation to return different values in Scenarios *0* and *0'*. This contradiction implies the claim. Let B_1 denote the first base object that p_n accesses.

To force p_n to access a second base object, the idea is to schedule other processes, before scheduling p_n, in such a way that they render the information in B_1 of little value to p_n. Consequently, later when p_n runs and accesses B_1, it will not learn enough to determine the response to its read operation on $\mathcal{O}$; thus, p_n is forced to access a second base object. The details are as follows. Let Scenario *1* depict an execution in which processes other than p_n take steps until some

process, say p_{i_1}, writes B_1.[1] If p_n runs after Scenario *1*, it accesses B_1 because, until accessing B_1, this scenario is indistinguishable to p_n from Scenario *0*. More significantly, we show below that, when running after Scenario *1*, p_n accesses some base object, besides B_1, before completing its first read operation on $\mathcal{O}$. To see this, consider another scenario, Scenario $1'$, which is the same as Scenario *1* with the following exception: just before p_{i_1} writes B_1, some process p_l other than p_{i_1} and p_n completes many increment operations on $\mathcal{O}$, and after this p_{i_1} takes a step and writes B_1. Clearly, B_1's value is the same at the end of Scenarios *1* and *1'*. Now extend each of Scenarios *1* and *1'* by letting p_n take steps. If p_n accesses *only* B_1, it is clear that the two scenarios would be indistinguishable to p_n. Yet, since many more increments operations completed in Scenario *1'* than in Scenario *1*, p_n's read operation on $\mathcal{O}$ must return different values in the two scenarios. It follows that, in Scenario *1* (and in Scenario *1'*), p_n must access a second base object before completing its read operation on $\mathcal{O}$. Let B_2 denote this base object.

To force p_n to access a third base object, we repeat the above trick and render the information in B_2 of little value to p_n. Specifically, consider Scenario *2* consisting of the following three execution fragments, taking place in that order: (i) the prefix of Scenario *1* (described above) up to, but excluding the write of B_1 by p_{i_1}, (ii) the steps of processes other than p_{i_1} and p_n until some process, say p_{i_2}, writes B_2, and (iii) the write of B_1 by p_{i_1}. If p_n were to run after Scenario *2*, it accesses B_1 and B_2 because, until accessing B_2, this scenario is indistinguishable to p_n from Scenario *1*. We claim that p_n accesses some base object, besides B_1 and B_2, before completing its first read operation on $\mathcal{O}$. The justification is as in the previous paragraph: if the claim is false, p_n cannot distinguish Scenario *2* from Scenario *2'*, where Scenario *2'* is similar to Scenario *2* except that some process p_l other than p_{i_1}, p_{i_2}, and p_n completes many increment operations on $\mathcal{O}$ just before the write steps of p_{i_2} and p_{i_1} on B_2 and B_1, respectively. This is a contradiction since p_n's read operation on $\mathcal{O}$ must return different values in Scenarios *2* and *2'*.

Repeating the above argument, we construct successively Scenarios $3, \ldots, n-2$ with the property that, if p_n runs alone after Scenario k, it accesses at least $k+1$ distinct base objects before completing its first read operation on $\mathcal{O}$. The lower bound of $n-1$ on the space and shared-access time complexity of the implementation is immediate from the existence of Scenario $n-2$. (We cannot proceed any further than Scenario $n-2$ because processes other than p_n are all already used up: they play the roles of $p_{i_1}, \ldots, p_{i_{n-2}}$ or as a process that does increment operations just before the write steps of $p_{i_1}, \ldots, p_{i_{n-2}}$.)

In summary, the crux is to ensure that p_n gets no useful information from $B_1, B_2, \ldots, B_{n-2}$; that is, $B_1, B_2, \ldots, B_{n-2}$ are rendered useless to p_n. This is accomplished by "using up" p_{i_j} to render B_j useless. This technique of using up one new process for each additional shared object to be rendered useless, in the context of space complexity lower bounds, was used earlier by Burns and Lynch [BL93].

We turn the above ideas into a rigorous inductive proof in Section 6.1.3. To understand the correspondence between that proof and the above informal argument, note that Scenario 2 described above has three parts: a schedule involving $p_1, \ldots, p_{n-1}$, followed by the write steps of p_{i_2} and p_{i_1} (on objects B_2 and B_1, respectively), followed by the steps of only p_n. More generally, if we extended the argument to Scenarios 3, 4, ..., Scenario k would consist of three

[1] It is possible that such an execution does not exist. To not obscure the basic intuition, we address this possibility only in the formal proof, presented in Section 6.1.3.

parts, where the first part is a schedule involving $p_1, \ldots, p_{n-1}$, the second part is the write steps of $p_{i_k}, p_{i_{k-1}}, \ldots, p_{i_1}$ on some base objects $B_k, B_{k-1}, \ldots, B_1$, and the third part is the steps of only p_n. Roughly speaking, these three parts of Scenario k' correspond to the schedules Λ, Σ, and Π, respectively, in Definition 1 in Section 6.1.2, and to the schedules Λ_k, Σ_k, and Π_k, respectively, of the proof in Section 6.1.3. We caution the reader that this correspondence is not exact, but is close enough to help the reader understand how the formal proof works.

6.1.2 Perturbable Types

The key property of the increment object exploited in the above proof is the following: it is possible to create a new scenario, by scheduling the steps of some process p_l immediately before that of $p_{i_k}, p_{i_{k-1}}, \ldots, p_{i_1}$, in such a way that p_n is forced to distinguish the new scenario from the older one. Below we state an abstract version of this property (which suffices for our proof technique to work), and call any type that has this property a *perturbable type*.

Definition 21 TYPE T IS PERTURBABLE FOR n PROCESSES, FOR INITIAL STATE s *if for every linearizable and solo-terminating randomized implementation of an object $\mathcal{O}$ of type T, initialized to s and shared by processes $p_1, \ldots, p_n$, there exists an assignment of operation sequences to input variables $oplist_1, \ldots, oplist_n$ such that the following statement holds:*

If Λ, Σ, and Π are any schedules that satisfy the following four conditions,

- PSET$(\Lambda) \subseteq \{p_1, \ldots, p_{n-1}\}$
- PSET(Σ) *is a proper subset of* $\{p_1, \ldots, p_{n-1}\}$ *and each process appears at most once in* Σ.
- PSET$(\Pi) = \{p_n\}$
- *In $\Lambda\Sigma\Pi$, p_n's first operation on $\mathcal{O}$ just completes and returns some response res.*

then, for some $p_l \in \{p_1, \ldots, p_{n-1}\} -$ PSET(Σ), there is a schedule $\gamma \in (\{p_l\} \times$ COINSPACE$)^*$ such that, in $\Lambda\gamma\Sigma\Pi$, either p_n's first operation on $\mathcal{O}$ does not complete or it returns a response different from res.

The key feature of a perturbable type can be informally explained as follows: Consider a perturbable object $\mathcal{O}$ in a configuration $\mathcal{C}$, with some pending operations by processes other than p_i. Let p_i be the only process to take steps from $\mathcal{C}$. Process p_i now invokes operation *op* and obtains a response x. x may be any of several values, depending on the unknown linearization order of the operations. However, the fact that $\mathcal{O}$ is perturbable implies that there exists a sequence S of operations, such that: Let $\mathcal{C}'$ be the configuration that results from $\mathcal{C}$, when the operations in S are executed sequentially to completion. (This definition of $\mathcal{C}'$ is only approximately true.) Let p_i be the only process to take steps from $\mathcal{C}'$. Process p_i now invokes operation *op* and obtains a response y. Then, y must necessarily be different from x, even though both x and y cannot be determined fully, due to the uncertain linearization orders.

There are schedules Λ_k, Σ_k, Π_k such that the following conditions hold:

1. Λ_k, $\Sigma_k \in (\{p_1, p_2, \ldots, p_{n-1}\} \times \text{COINSPACE})^*$ and $\Pi_k \in (\{p_n\} \times \text{COINSPACE})^*$. That is, Λ_k and Σ_k do not contain p_n and Π_k contains no process other than p_n.

2. $|\Sigma_k| = |\text{PSET}(\Sigma_k)| = k$. That is, k distinct processes take one step each in Σ_k.

3. In $\Lambda_k\Sigma_k\Pi_k$, p_n accesses exactly k distinct base objects and p_n's first operation on $\mathcal{O}$ has either not completed or just completed.

4. Let $\mathcal{S}_k$ be the set of base objects that p_n accesses in $\Lambda_k\Sigma_k\Pi_k$. Let $\mathcal{P}_k = \{p_1, p_2, \ldots, p_{n-1}\} - \text{PSET}(\Sigma_k)$ and γ be any schedule in $(\mathcal{P}_k \times \text{COINSPACE})^*$. Then, $\Lambda_k\Sigma_k \approx_{\mathcal{S}_k} \Lambda_k\gamma\Sigma_k$.

Figure 6.1: Statement S_k

6.1.3 The Main Result

We prove that the space complexity of any randomized solo-terminating implementation and the shared-access time complexity of any deterministic solo-terminating implementation of any perturbable object, shared by n processes, are both at least $n-1$ if base objects are restricted to be (any combination of) resettable consensus objects and historyless objects, such as registers, test&set objects, and swap registers.

Theorem 8 *Suppose that type T is perturbable for n processes, for some initial state s. Consider any randomized implementation of an object of type T, initialized to s and shared by processes $p_1, \ldots, p_n$, from resettable consensus objects and historyless objects. If the implementation is linearizable and solo-terminating:*

1. *Its space complexity is at least $n-1$.*

2. *If the implementation is deterministic, its solo-termination shared-access time complexity is at least $n-1$.*

Proof Let $\mathcal{O}$ be the implemented object, and C_0 be the initial configuration of $(p_1, \ldots, p_n; \mathcal{O})$ obtained by assigning to $\textit{op-list}_1, \ldots, \textit{op-list}_n$ the operation sequences mentioned in Definition 21. The crux of the proof lies in the following claim: For all k, $0 \le k \le n-1$, Statement S_k, presented in Figure 6.1, is true. We prove this claim by induction. Below, we let $S_k : j$ denote the jth part of Statement S_k.
Base case: Let $\Lambda_0 = \Sigma_0 = \Pi_0 = \epsilon$ (ϵ is the empty sequence). It is easy to verify that all of S_0 : 1-4 are true. Hence, we have the base case.
Induction step: Suppose $0 \le k \le n-2$ and S_k is true. Let $\Lambda_k, \Sigma_k, \Pi_k$ be so defined as to make Statement S_k true. Let $\mathcal{S}_k$ denote the set of base objects that p_n accesses in $\Lambda_k\Sigma_k\Pi_k$, and $\mathcal{P}_k$ denote $\{p_1, p_2, \ldots, p_{n-1}\} - \text{PSET}(\Sigma_k)$. We now show that S_{k+1} is true.

By $S_k:3$, in $\Lambda_k\Sigma_k\Pi_k$, p_n's first operation on $\mathcal{O}$ has either not completed or just completed. Let $\pi \in (\{p_n\} \times \text{COINSPACE})^*$ be such that, in $\Lambda_k\Sigma_k\Pi_k\pi$, p_n just completes its first operation on $\mathcal{O}$, returning some value. Since the implementation is solo-terminating, π exists.

Claim 4 *$\pi \neq \epsilon$ and in $\Lambda_k\Sigma_k\Pi_k\pi$, p_n accesses a base object not in $\mathcal{S}_k$.*

Proof Suppose the claim is false. Recall that $\mathcal{P}_k = \{p_1, p_2, \ldots, p_{n-1}\} - \text{PSET}(\Sigma_k)$. Since $|\text{PSET}(\Sigma_k)| = k$ and $k \leq n-2$, $\mathcal{P}_k$ is non-empty. For all $p_l \in \mathcal{P}_k$ and $\gamma \in (\{p_l\} \times \text{COINSPACE})^*$, we assert that $\Lambda_k\Sigma_k\Pi_k\pi \approx_{p_n} \Lambda_k\gamma\Sigma_k\Pi_k\pi$. This assertion follows from the facts below: (i) $\Lambda_k\Sigma_k \approx_{p_n} \Lambda_k\gamma\Sigma_k$ (since the schedules $\Lambda_k\Sigma_k$ and $\Lambda_k\gamma\Sigma_k$ do not contain p_n), (ii) $\Lambda_k\Sigma_k \approx_{\mathcal{S}_k} \Lambda_k\gamma\Sigma_k$ (by $S_k:4$), (iii) the schedule $\Pi_k\pi$ contains only p_n, and (iv) the only base objects accessed by p_n in $\Lambda_k\Sigma_k\Pi_k\pi$ are the ones in $\mathcal{S}_k$ (by our assumption that Claim 4 is false).

The above assertion, together with the definition of π, implies that p_n's first operation on $\mathcal{O}$ completes and returns the same response in $\Lambda_k\gamma\Sigma_k\Pi_k\pi$ as in $\Lambda_k\Sigma_k\Pi_k\pi$. This contradicts Definition 21 (to see the contradiction, substitute Λ, Σ, Π in the definition with $\Lambda_k, \Sigma_k, \Pi_k\pi$, respectively, and note that the conditions in the definition hold because of the induction hypothesis). Hence, we have Claim 4. □

Definition 22 *Define π_{k+1}, B_{k+1}, and Π_{k+1} as follows:*

- *π_{k+1} is the shortest prefix of π such that, in $\Lambda_k\Sigma_k\Pi_k\pi_{k+1}$, p_n accesses a base object not in $\mathcal{S}_k$ (by Claim 4, π_{k+1} exists).*
- *B_{k+1} is the unique base object not in $\mathcal{S}_k$ that p_n accesses in $\Lambda_k\Sigma_k\Pi_k\pi_{k+1}$.*
- *$\Pi_{k+1} = \Pi_k\pi_{k+1}$.*

Claim 5 *$\Pi_{k+1} \in (\{p_n\} \times \text{COINSPACE})^*$.*

Proof Since Π_k and π_{k+1} are both from $(\{p_n\} \times \text{COINSPACE})^*$, we have $\Pi_{k+1} \in (\{p_n\} \times \text{COINSPACE})^*$. □

Claim 6 *There exist $\lambda_{k+1} \in (\mathcal{P}_k \times \text{COINSPACE})^*$ and $[p_{i_{k+1}}, t_{k+1}] \in \mathcal{P}_k \times \text{COINSPACE}$ such that, for all $\gamma \in ((\mathcal{P}_k - \{p_{i_{k+1}}\}) \times \text{COINSPACE})^*$, $\Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}] \approx_{B_{k+1}} \Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]$.*

Proof The following observation will be used many times in the proof:

> Observation **O1**: For all $\gamma \in ((\mathcal{P}_k - \{p_{i_{k+1}}\}) \times \text{COINSPACE})^*$, Process $p_{i_{k+1}}$ accesses the same base object and applies the same operation in the last step of $\Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]$ as in the last step of $\Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]$.
>
> This observation follows from the fact that γ does not contain $p_{i_{k+1}}$.

In the following, we pick λ_{k+1} and $[p_{i_{k+1}}, t_{k+1}]$ based on the type of B_{k+1}, and in each case prove that our choice of λ_{k+1} and $[p_{i_{k+1}}, t_{k+1}]$ satisfies Claim 6. In the rest of the proof, γ denotes an arbitrary schedule in $((\mathcal{P}_k - \{p_{i_{k+1}}\}) \times \text{COINSPACE})^*$.

Case 1 B_{k+1} is a historyless object.

Subcase 1a There is some non-empty schedule $\lambda \in (\mathcal{P}_k \times \text{COINSPACE})^*$ such that the last step in $\Lambda_k\lambda$ is a nontrivial operation on B_{k+1}.

Define λ_{k+1} and $[p_{i_{k+1}}, t_{k+1}]$ so that $\lambda = \lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]$.

By **O1**, $p_{i_{k+1}}$ performs the same nontrivial operation on B_{k+1} in the last step of $\Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]$ as in the last step of $\Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]$. This, together with Proposition 1, gives Claim 6.

Subcase 1b There is no such λ.

Define λ_{k+1} to be ϵ and $[p_{i_{k+1}}, t_{k+1}]$ to be any element of $\mathcal{P}_k \times \text{COINSPACE}$.

It follows from the subcase in consideration that no nontrivial operation is performed on B_{k+1} in the last $|\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]|$ steps of $\Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]$ and in the last $|\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]|$ steps of $\Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]$. Therefore, by Proposition 1,
$\Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}] \approx_{B_{k+1}} \Lambda_k \approx_{B_{k+1}} \Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]$. Hence, we have Claim 6.

Case 2 B_{k+1} is a resettable consensus object.

Subcase 2a There is some non-empty schedule $\lambda \in (\mathcal{P}_k \times \text{COINSPACE})^*$ such that the last step in $\Lambda_k\lambda$ is a reset operation on B_{k+1}.

Define λ_{k+1} and $[p_{i_{k+1}}, t_{k+1}]$ so that $\lambda = \lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]$.

By **O1**, $p_{i_{k+1}}$ performs a reset on B_{k+1} in the last step of $\Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]$, just as it does in the last step of $\Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]$. Hence, we have Claim 6.

Subcase 2b There is no such λ. However, there is some non-empty schedule $\lambda' \in (\mathcal{P}_k \times \text{COINSPACE})^*$ such that the last step in $\Lambda_k\lambda'$ is a propose operation on B_{k+1}.

Define λ_{k+1} to be λ' and $[p_{i_{k+1}}, t_{k+1}]$ to be any element of $\mathcal{P}_k \times \text{COINSPACE}$.

Let σ be the state of B_{k+1} at the end of $\Lambda_k\lambda_{k+1}$. Since Subcase 2a is not applicable, it follows that B_{k+1} is not reset in the last $|\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]|$ steps of $\Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]$ and in the last $|\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]|$ steps of $\Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]$. Thus, B_{k+1}'s state is σ at the end of $\Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]$ and at the end of $\Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]$. Hence, we have Claim 6.

Subcase 2c Neither λ nor λ' exists.

Define λ_{k+1} to be ϵ and $[p_{i_{k+1}}, t_{k+1}]$ to be any element of $\mathcal{P}_k \times \text{COINSPACE}$.

It follows from the subcase under consideration that no reset or propose operation is performed on B_{k+1} in the last $|\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]|$ steps of $\Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]$ and in the last $|\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]|$ steps of $\Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]$. Therefore, B_{k+1}'s state at the end of $\Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]$ is the same as its state at the end of $\Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]$. Hence, we have Claim 6.

This completes the proof of Claim 6. □

Definition 23 *Let λ_{k+1} and $[p_{i_{k+1}}, t_{k+1}]$ be as in Claim 6. Define Λ_{k+1} and Σ_{k+1} as follows:*

- $\Lambda_{k+1} = \Lambda_k\lambda_{k+1}$

- $\Sigma_{k+1} = [p_{i_{k+1}}, t_{k+1}]\Sigma_k$

Claim 7 $\Lambda_{k+1}, \Sigma_{k+1} \in (\{p_1, p_2, \ldots, p_{n-1}\} \times \text{COINSPACE})^*$.

Proof By definition, $\lambda_{k+1} \in (\mathcal{P}_k \times \text{COINSPACE})^*$ and $[p_{i_{k+1}}, t_{k+1}] \in \mathcal{P}_k \times \text{COINSPACE}$, where $\mathcal{P}_k = \{p_1, p_2, \ldots, p_{n-1}\} - \text{PSET}(\Sigma_k)$. This, together with $S_k : 1$, implies the claim. □

Claim 8 $|\Sigma_{k+1}| = |\text{PSET}(\Sigma_{k+1})| = k + 1$.

Proof This claim follows from the definition of Σ_{k+1} as $[p_{i_{k+1}}, t_{k+1}]\Sigma_k$ and the following two facts: (i) $|\Sigma_k| = |\text{PSET}(\Sigma_k)| = k$ (by $S_k : 2$), and (ii)$[p_{i_{k+1}}, t_{k+1}] \in \mathcal{P}_k \times \text{COINSPACE}$ and $\mathcal{P}_k$ does not include any process from $\text{PSET}(\Sigma_k)$. □

Claim 9 *Let* $\mathcal{P}_{k+1} = \{p_1, p_2, \ldots, p_{n-1}\} - \text{PSET}(\Sigma_{k+1})$. *Let* γ *be any schedule from* $(\mathcal{P}_{k+1} \times \text{COINSPACE})^*$. *Then we have:*

1. $\Lambda_{k+1}[p_{i_{k+1}}, t_{k+1}] \approx_{B_{k+1}} \Lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]$.
2. $\Lambda_{k+1}\Sigma_{k+1} \approx_{B_{k+1}} \Lambda_{k+1}\gamma\Sigma_{k+1}$.
3. $\Lambda_{k+1}\Sigma_{k+1} \approx_{\mathcal{S}_k} \Lambda_k\Sigma_k \approx_{\mathcal{S}_k} \Lambda_{k+1}\gamma\Sigma_{k+1}$.

Proof Part (a) of this claim is a rephrasing of Claim 6. The proof of the other two parts of this claim will use Observation **O1**, stated earlier in the proof of Claim 6.

By construction, $p_{i_{k+1}} \notin \text{PSET}(\Sigma_k)$. By definition of γ, $\text{PSET}(\gamma) \cap \text{PSET}(\Sigma_k) = \emptyset$. It follows that $\Lambda_{k+1}[p_{i_{k+1}}, t_{k+1}] \approx_{\text{PSET}(\Sigma_k)} \Lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]$. From this, the fact that each process in $\text{PSET}(\Sigma_k)$ appears only once in Σ_k and $|\Sigma_k| = k$, we conclude that the sequence of base objects accessed and the operations applied in the last k steps of $\Lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]\Sigma_k$ are identical to the sequence of base objects accessed and the operations applied in the last k steps of $\Lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]\Sigma_k$. This, together with part (a) of the claim, implies part (b).

It follows from the induction hypothesis $S_k\!:\!4$ that $\Lambda_k\lambda_{k+1}[p_{i_{k+1}}, t_{k+1}]\Sigma_k \approx_{\mathcal{S}_k} \Lambda_k\Sigma_k \approx_{\mathcal{S}_k} \Lambda_k\lambda_{k+1}\gamma[p_{i_{k+1}}, t_{k+1}]\Sigma_k$. This, together with prior definitions of Λ_{k+1} as $\Lambda_k\lambda_{k+1}$ and Σ_{k+1} as $[p_{i_{k+1}}, t_{k+1}]\Sigma_k$, gives part (c) of the claim. □

Claim 10

1. *Let* $\mathcal{S}_{k+1}$ *be the set of base objects that* p_n *accesses in* $\Lambda_{k+1}\Sigma_{k+1}\Pi_{k+1}$. *Then,* $\mathcal{S}_{k+1} = \mathcal{S}_k \cup \{B_{k+1}\}$ *and* $|\mathcal{S}_{k+1}| = k + 1$.
2. *In* $\Lambda_{k+1}\Sigma_{k+1}\Pi_{k+1}$, p_n*'s first operation on* $\mathcal{O}$ *has either not completed or just completed.*

Proof We make the following observations: (1) $\Lambda_k\Sigma_k \approx_{p_n} \Lambda_{k+1}\Sigma_{k+1}$ (since neither $\Lambda_k\Sigma_k$ nor $\Lambda_{k+1}\Sigma_{k+1}$ contains p_n), (2) $\Lambda_k\Sigma_k \approx_{\mathcal{S}_k} \Lambda_{k+1}\Sigma_{k+1}$ (this is part (c) of Claim 9), (3) By definition of $\mathcal{S}_k$, $\mathcal{S}_k$ is exactly the set of base objects that p_n accesses in $\Lambda_k\Sigma_k\Pi_k$, and (4) By definition of π_{k+1}, in $\Lambda_k\Sigma_k\Pi_k\pi_{k+1}$, it is only in the last step that p_n accesses a base object not in $\mathcal{S}_k$ (this base object is B_{k+1}), and p_n's first operation on $\mathcal{O}$ has either not completed or just completed.

These observations imply Part (b) of the claim and that $\mathcal{S}_{k+1} = \mathcal{S}_k \cup \{B_{k+1}\}$. This, together with $|\mathcal{S}_k| = k$ (by induction hypothesis), implies $|\mathcal{S}_{k+1}| = k + 1$.

□

We have proved all four parts of Statement S_{k+1}: Part 1 in Claims 7 and 5, Part 2 in Claim 8, Part 3 in Claim 10, and Part 4 follows from parts (b) and (c) of Claim 9 and Claim 7(a). This completes the induction step and hence, the proof of Statement S_k, for all $0 \leq k \leq n-1$.

We now proceed to prove Theorem 8. The first part of the theorem is immediate from Part 3 of Statement S_{n-1}. To obtain the second part of the theorem, observe that a deterministic implementation can be viewed as a randomized implementation for which COINSPACE is a singleton set. Since Statement S_k ($0 \leq k \leq n-1$) is proved for any non-empty countable COINSPACE, Statement S_{n-1} is true for any deterministic implementation. By Part 3 of Statement S_{n-1}, in $\Lambda_{n-1}\Sigma_{n-1}\Pi_{n-1}$, p_n accesses $n-1$ base objects and has either not completed or just completed its first operation on $\mathcal{O}$. This implies that the solo-termination time complexity is at least $n-1$. Hence, we have the theorem. □

6.2 Examples of Perturbable Types

We show that the following common types of objects are perturbable for n processes: modulo k counter for any $k \geq 2n$, increment object, fetch&add, k-valued compare&swap for any $k \geq n$, LL/SC bit, and single writer snapshot. It follows from Theorem 8 that the space complexity of a randomized implementation or the shared-access time complexity of a deterministic implementation of any of these objects from resettable consensus objects and historyless objects is at least $n-1$.

6.2.1 Modulo Counter and Related Objects

A *modulo k counter* supports *increment* and *read* operations. The states are $0, 1, \ldots, k-1$. The *increment* operation adds 1 to the state (modulo k) and returns *ack*. The *read* operation returns the state, without affecting it. The following proposition is immediate from the linearizability requirement.

Proposition 2 *Let $\mathcal{O}$ be a modulo k counter, initialized to 0. Let E be a finite execution of $(p_1, \ldots, p_n; \mathcal{O})$ such that in the configuration C at the end of E, process p_n has no pending operation on $\mathcal{O}$. Suppose p_n runs alone from C and performs a read operation on $\mathcal{O}$. If the number of completed increments in E is at least v and the sum in E of the number of completed increments and the number of pending increments is at most v', then the value returned by the read of p_n is in the range $[v, v'] \bmod k$.*

Lemma 81 *For all $k \geq 2n$, modulo k counter is perturbable for n processes, for any initial state.*

Proof Without loss of generality, we prove the lemma for initial state 0. Consider any linearizable and solo-terminating randomized implementation of a modulo k counter $\mathcal{O}$, initialized to 0 and shared by processes $p_1, \ldots, p_n$. For $1 \leq i \leq n-1$, let *op-list*$_i$ be an infinite sequence of

increment operations, and $op\text{-}list_n$ be an infinite sequence of *read* operations. Let Λ, Σ, and Π be any schedules that satisfy the four conditions listed in Definition 21.

Let p_l be any process in $\{p_1, \ldots, p_{n-1}\} - \text{PSET}(\Sigma)$, and $\gamma \in (\{p_l\} \times \text{COINSPACE})^*$ be the shortest schedule such that there are exactly n more completed increment operations on $\mathcal{O}$ in $\Lambda\gamma$ than in Λ. Since the implementation is solo-terminating, γ exists. We now make the following observations:

1. If a process completes an increment on $\mathcal{O}$ in the last $|\Sigma|$ steps of $\Lambda\Sigma$, then it has no pending increment on $\mathcal{O}$ in $\Lambda\Sigma$. Furthermore, no process completes more than one increment on $\mathcal{O}$ in the last $|\Sigma|$ steps of $\Lambda\Sigma$.

   This follows from the fact that each process appears at most once in the schedule Σ.

2. For any execution E, let $\text{NP}(E)$ denote the number of pending increment operations on $\mathcal{O}$ in E. The sum of $\text{NP}(\Lambda\Sigma)$ and the number of increments that completed in the last $|\Sigma|$ steps of $\Lambda\Sigma$ is at most $n-1$.

   This follows from Observation 1 and the fact that $\text{PSET}(\Lambda\Sigma) \subseteq \{p_1, \ldots, p_{n-1}\}$.

3. The sum of $\text{NP}(\Lambda\gamma\Sigma)$ and the number of increments that completed in the last $|\Sigma|$ steps of $\Lambda\gamma\Sigma$ is at most $n-1$.

   This also follows from Observation 1 and the fact that $\text{PSET}(\Lambda\gamma\Sigma) \subseteq \{p_1, \ldots, p_{n-1}\}$.

4. For any execution E, let $\text{NC}(E)$ denote the number of completed increment operations on $\mathcal{O}$ in E. Let $\text{NC}(\Lambda) = v$. In $\Lambda\Sigma\Pi$, the value res, which p_n's first operation on $\mathcal{O}$ (which is a read) returns, is in the range $[v, v+n-1] \bmod k$.

   This follows from Proposition 2 and the following two chains of inequalities:

   $$\text{NC}(\Lambda\Sigma) \geq \text{NC}(\Lambda) = v$$

   $$\begin{aligned} &\text{NC}(\Lambda\Sigma) + \text{NP}(\Lambda\Sigma) \\ &= \text{NC}(\Lambda) + \text{NP}(\Lambda\Sigma) + \\ &\quad \text{number of increments that completed in} \\ &\quad \text{the last } |\Sigma| \text{ steps of } \Lambda\Sigma \\ &\leq v + n - 1 \qquad \text{(by Observation 2)} \end{aligned}$$

5. In $\Lambda\gamma\Sigma\Pi$, if p_n's first operation on $\mathcal{O}$ completes, it returns a value in the range $[v+n, v+2n-1] \bmod k$.

   This follows from Proposition 2 and the following two chains of inequalities:

   $$\text{NC}(\Lambda\gamma\Sigma) \geq \text{NC}(\Lambda\gamma) = v + n$$

   $$\begin{aligned} &\text{NC}(\Lambda\gamma\Sigma) + \text{NP}(\Lambda\gamma\Sigma) \\ &= \text{NC}(\Lambda\gamma) + \text{NP}(\Lambda\gamma\Sigma) + \\ &\quad \text{number of increments that completed in} \\ &\quad \text{the last } |\Sigma| \text{ steps of } \Lambda\gamma\Sigma \\ &\leq (v+n) + (n-1) \qquad \text{(by Observation 3)} \end{aligned}$$

Since $k \geq 2n$, the range $[v, v+n-1] \bmod k$ and the range $[v+n, v+2n-1] \bmod k$ are disjoint. This, together with Observations 4 and 5, implies Lemma 81. □

An *increment object* is a special case of a modulo k counter, for $k = \infty$. Since the finiteness of k is not used in the proofs of Proposition 2 or Lemma 81, we have the following result:

Lemma 82 *An increment object is perturbable for n processes, for any initial state.*

A *fetch&add object* supports the operation fetch&add(v), for any integer v. The states are integers. The fetch&add(v) operation adds v to the state and returns the previous state. Proceeding analogously as in the proof of Lemma 81, with $k = \infty$ and the operations *increment*, *read* replaced by *fetch&add*(1), *fetch&add*(0), we obtain the following lemma.

Lemma 83 *A fetch&add object is perturbable for n processes, for any initial state.*

6.2.2 Compare&Swap

A *k-valued compare&swap object* supports the operations *read* and *c&s*(u, v), for all $u, v \in \{1, 2, \ldots, k\}$. The states are $1, 2, \ldots, k$. The effect of *c&s*(u, v) depends on whether or not the state is u: if the state is u, *c&s*(u, v) changes the state to v and returns *true*; otherwise it returns *false* without affecting the state. We say a compare&swap operation is *successful* if it returns *true*.

Proposition 3 *Let C be any reachable configuration of $(p_1, \ldots, p_n; \mathcal{O})$, where $\mathcal{O}$ is a k-valued compare&swap object. Suppose that process p_l has no pending operations on $\mathcal{O}$ in C. For any $w \in \{1, 2, \ldots, k\}$, if p_l runs alone from C, completing the sequence of operations read, c&s$(1, w)$, c&s$(2, w)$, ..., c&s(k, w), then one of the following is true:*

1. *One of the c&s operations of p_l returns true.*
2. *Some operation on $\mathcal{O}$ that was pending in C is linearized after the read and before the last c&s operation of p_l.*

Proof Let v be the value returned by the read operation of p_l. Suppose that Statement 1 in the Proposition is false. Since *c&s*(v, w), which is one of the n *c&s* operations that p_l performed following the read, did not return true, some pending operation must have taken effect after the read and before the *c&s*(v, w). □

Proposition 4 *Let C be any reachable configuration of $(p_1, \ldots, p_n; \mathcal{O})$, where $\mathcal{O}$ is a k-valued compare&swap object. Suppose that process p_l has no pending operations on $\mathcal{O}$ in C. Let $w \in \{1, 2, \ldots, k\}$ and suppose that p_l runs alone from C, completing the following sequence of operations n times: read, c&s$(1, w)$, c&s$(2, w)$, ..., c&s(k, w). Then, at least one of the c&s operations returns true.*

Proof Follows by successive application of Proposition 3 and the observation that there can be at most $n - 1$ pending operations on $\mathcal{O}$ in C. □

Lemma 84 *For all $k \geq n$, k-valued compare&swap object is perturbable for n processes, for any initial state.*

Proof Consider any linearizable and solo-terminating randomized implementation of a k-valued compare&swap object $\mathcal{O}$, initialized to any value and shared by processes $p_1, \ldots, p_n$. For any $1 \leq j \leq k$, let α_j denote the operation sequence *read*, $c\&s(1,j), c\&s(2,j), \ldots, c\&s(k,j)$. Let β denote the operation sequence $\alpha_1^n \alpha_2^n \ldots \alpha_k^n$, where α_i^m denotes the sequence α_i repeated m times. Thus, $|\alpha_j| = k+1$ and $|\beta| = nk(k+1)$. For all $1 \leq i \leq n-1$, initialize the input variable *op-list*$_i$ to the infinite sequence $\beta\beta\beta\ldots$, and initialize *op-list*$_n$ to the infinite sequence of *read* operations. Let Λ, Σ, and Π be any schedules that satisfy the four conditions listed in Definition 21.

Let p_l be any process in $\{p_1, \ldots, p_{n-1}\} - \text{PSET}(\Sigma)$. If p_l has any pending operation on $\mathcal{O}$ at the end of Λ, let $\gamma' \in (\{p_l\} \times \text{COINSPACE})^*$ be such that p_l just completes that operation in $\Lambda\gamma'$. Otherwise let $\gamma' = \epsilon$. Thus, at the end of $\Lambda\gamma'$, p_l has no pending operation on $\mathcal{O}$, and any pending operations have to be from processes in $\{p_1, \ldots, p_{n-1}\} - \{p_l\}$. Let $P \subseteq \{p_1, \ldots, p_{n-1}\} - \{p_l\}$ be the set of processes that have pending operations on $\mathcal{O}$ at the end of $\Lambda\gamma'$.

Let Q be the set of processes that initiate a new operation on $\mathcal{O}$ in the last $|\Sigma|$ steps of $\Lambda\gamma'\Sigma$. Since $p_l \in \{p_1, \ldots, p_{n-1}\} - \text{PSET}(\Sigma)$, we have $Q \subseteq \{p_1, \ldots, p_{n-1}\} - \{p_l\}$. Furthermore, since each process appears at most once in Σ, if a process has a pending operation in $\Lambda\gamma'$ then that process cannot initiate a new operation on $\mathcal{O}$ in the last $|\Sigma|$ steps of $\Lambda\gamma'\Sigma$. In other words, $P \cap Q = \emptyset$. From this and the fact $P, Q \subseteq \{p_1, \ldots, p_{n-1}\} - \{p_l\}$, we have $|P| + |Q| \leq n-2$. That is, the sum of the number of pending operations on $\mathcal{O}$ in $\Lambda\gamma'$ and the number of operations on $\mathcal{O}$ initiated in the last $|\Sigma|$ steps of $\Lambda\gamma'\Sigma$ is at most $n-2$. Let V be the set of all v such that a $c\&s(v,*)$ operation[2] on $\mathcal{O}$ is either pending in $\Lambda\gamma'$ or initiated in the last $|\Sigma|$ steps of $\Lambda\gamma'\Sigma$. From the above, $|V| \leq n-2$.

Recall from (the fourth condition in) Definition 21 that *res* is the value returned by p_n's first operation on $\mathcal{O}$ in $\Lambda\Sigma\Pi$. Let $w \in \{1, 2, \ldots, n\}$ be such that $w \notin V$ and $w \neq res$. Let $\gamma'' \in (\{p_l\} \times \text{COINSPACE})^*$ be the shortest schedule such that, in $\Lambda\gamma'\gamma''$, we have: (i) p_l has no pending operations, (ii) there are at least $n(k+1)$ completed operations on $\mathcal{O}$ (by p_l) in the last $|\gamma''|$ steps, and (iii) the sequence of $n(k+1)$ most recent completed operations of p_l on $\mathcal{O}$ is α_w^n. The definition of *op-list*$_l$ and the fact that the implementation is solo-terminating imply that γ'' exists. We now make the following observations:

1. In $\Lambda\gamma'\gamma''$, the most recent $n(k+1)$ operations of p_l on $\mathcal{O}$ includes a *successful* compare&swap operation of the form $c\&s(*,w)$. (Let OP denote any such operation.)

   Proof In $\Lambda\gamma'\gamma''$, the most recent $n(k+1)$ operations of p_l on $\mathcal{O}$ are the operations in α_w^n. All of the compare&swap operations in α_w^n are of the form $c\&s(*,w)$ and, by Proposition 4, at least one of these succeeds. □

2. Consider any linearization of $\Lambda\gamma'\gamma''\Sigma$. If OP$'$ is a successful compare&swap operation that is linearized after OP, then OP$'$ must be of the form $c\&s(*,w)$.

   Proof We prove this assertion by contradiction. Let OP$'$ be the first successful compare&swap operation that is of the form $c\&s(*,x)$, for some $x \neq w$, to be linearized after

[2] An asterisk in a field indicates that we do not care what the value of that field is.

OP. Since OP is successful and each successful compare&swap operation that is linearized after OP and before OP$'$ is of the form $c\&s(*, w)$, the value of the object is w immediately before OP$'$.

There are three cases to consider: (i) OP$'$ is an operation from p_l that follows OP, (ii) OP$'$ is an operation which is pending in $\Lambda\gamma'$, or (iii) OP$'$ is an operation which is initiated in the last $|\Sigma|$ steps of $\Lambda\gamma'\gamma''\Sigma$. In Case (i), by definition of γ'' and OP, OP$'$ is of the form $c\&s(*, w)$, a contradiction. In Cases (ii) and (iii), by definitions of V and w, OP$' = c\&s(v, *)$ for some $v \neq w$. Since the value of the object is w immediately before OP$'$, the fact $v \neq w$ implies that OP$' = c\&s(v, *)$ cannot be successful, a contradiction. □

3. In $\Lambda\gamma'\gamma''\Sigma\Pi$, if p_n's first operation on $\mathcal{O}$ (which is a read operation) completes, it returns a value different from *res*.

   Proof Since OP is successful and is of the form $c\&s(*, w)$, the value of $\mathcal{O}$ immediately after OP is w. By the previous observation, in $\Lambda\gamma'\gamma''\Sigma$, every successful compare&swap linearized after OP is also of the form $c\&s(*, w)$.

   Therefore, in $\Lambda\gamma'\gamma''\Sigma\Pi$, if p_n's first operation on $\mathcal{O}$ (which is a read operation) completes, it returns w. But w, by definition, is different from *res*. Hence, we have the observation. □

Lemma 84 is immediate from the last observation. □

6.2.3 LL/SC Bit

An *n-process load-link store-conditional bit* (n-process LL/SC bit) supports the operations *LL* and *SC*(b), for $b = 0, 1$. The states are pairs (v, S), for all $v \in \{0, 1\}$ and $S \subseteq \{1, 2, \ldots, n\}$. The operation *LL* from process p_i, when applied in state (v, S), returns v and changes the state to (v, S'), where $S' = S \cup \{i\}$. The operation *SC*(b) from process p_i, when applied in state (v, S), has the following effect: if $i \in S$, the state changes to $(b, \emptyset)$ and *true* is returned; otherwise the state is not affected and *false* is returned. We say an *SC* operation is *successful* if it returns *true*.

Proposition 5 *Let C be any reachable configuration of $(p_1, \ldots, p_n; \mathcal{O})$, where $\mathcal{O}$ is an n-process LL/SC bit. Suppose that process p_l has no pending operations on $\mathcal{O}$ in C. If p_l runs alone from C, completing an LL operation and then an SC(b) operation (for any $b \in \{0, 1\}$), then one of the following is true:*

1. *The SC(b) operation of p_l returns true.*

2. *Some SC operation on $\mathcal{O}$ that was pending in C is linearized after the LL and before the SC(b) of p_l.*

Proof Follows from the specification of n-process LL/SC bit. □

Proposition 6 *Let C be any reachable configuration of $(p_1, \ldots, p_n; \mathcal{O})$, where $\mathcal{O}$ is an n-process LL/SC bit. Suppose that process p_l has no pending operations on $\mathcal{O}$ in C. For $b \in \{0, 1\}$, suppose further that p_l runs alone from C, completing the following sequence of operations n times: LL, SC(b). Then, at least one of the SC(b) operations returns true.*

Proof Follows by repeated application of Proposition 5 and the observation that there can be at most $n-1$ pending operations on $\mathcal{O}$ in C. □

Lemma 85 *LL/SC bit is perturbable for n processes, for any initial state.*

Proof Consider any linearizable and solo-terminating randomized implementation of an LL/SC bit $\mathcal{O}$, initialized to any value and shared by processes $p_1, \ldots, p_n$. For $j \in \{0,1\}$, let α_j denote the sequence *LL*, $SC(j)$, *LL*, $SC(j)$, ..., *LL*, $SC(j)$ that has a total of $2n$ operations (n *LL* operations and n $SC(j)$ operations). For all $1 \le i \le n-1$, initialize the input variable $op\text{-}list_i$ to the infinite sequence $\alpha_0, \alpha_1, \alpha_0, \alpha_1, \alpha_0, \alpha_1, \ldots$ and initialize $op\text{-}list_n$ to the infinite sequence of *LL* operations. Let Λ, Σ, and Π be any schedules that satisfy the four conditions listed in Definition 21.

Let p_l be any process in $\{p_1, \ldots, p_{n-1}\} - \text{PSET}(\Sigma)$. If p_l has any pending operation on $\mathcal{O}$ at the end of Λ, let $\gamma' \in (\{p_l\} \times \text{COINSPACE})^*$ be such that p_l just completes that operation in $\Lambda\gamma'$. Otherwise let $\gamma' = \epsilon$. Thus, at the end of $\Lambda\gamma'$, p_l has no pending operation on $\mathcal{O}$, but other processes may. Any such pending operations have to be from processes in $\{p_1, \ldots, p_{n-1}\} - \{p_l\}$.

Recall from (the fourth condition in) Definition 21 that res is the value returned by p_n's first operation on $\mathcal{O}$ in $\Lambda\Sigma\Pi$. Let $w = 1 - res$. Let $\gamma'' \in (\{p_l\} \times \text{COINSPACE})^*$ be the shortest schedule such that, in $\Lambda\gamma'\gamma''$, we have: (i) p_l has no pending operations, (ii) there are at least $2n$ completed operations on $\mathcal{O}$ (by p_l) in the last $|\gamma''|$ steps, and (iii) the sequence of $2n$ most recent operations of p_l on $\mathcal{O}$ is α_w. The definition of $op\text{-}list_l$ and the fact that the implementation is solo-terminating imply that γ'' exists. We now make the following observations:

1. In $\Lambda\gamma'\gamma''$, the most recent $2n$ operations of p_l on $\mathcal{O}$ includes a successful $SC(w)$ operation. (Let OP denote any such operation.)

   Proof Consider the sequence α_w of the $2n$ most recent (alternating *LL* and $SC(w)$) operations of p_l on $\mathcal{O}$. By Proposition 6, at least one of these $SC(w)$ operations succeeds. □

2. Consider any linearization of $\Lambda\gamma'\gamma''\Sigma$. Let $\text{OP}' = SC(v)$ be any successful operation that is linearized after OP. Then $v = w$.

   Proof If OP′ is linearized after OP, there are three cases to consider: (i) OP′ is an operation from p_l that follows OP, (ii) OP′ is an operation which is pending in $\Lambda\gamma'$, or (iii) OP′ is an operation which is initiated in the last $|\Sigma|$ steps of $\Lambda\gamma'\gamma''\Sigma$. In the following we show that the observation holds in all cases.

   <u>Case (i): OP′ is an operation from p_l that follows OP</u>

   Since $p_l \in \{p_1, \ldots, p_{n-1}\} - \text{PSET}(\Sigma)$, p_l has no step in the last $|\Sigma|$ steps of $\Lambda\gamma'\gamma''\Sigma$. From this and the definition of γ'', the $2n$ most recent operations from p_l in $\Lambda\gamma'\gamma''\Sigma$ are *LL*, $SC(w)$, *LL*, $SC(w)$, ..., *LL*, $SC(w)$. By definition, OP is one of these $2n$ operations. Thus, if $\text{OP}' = SC(v)$ is an *SC* operation from p_l that follows OP, then v must equal w.

   <u>Cases (ii) and (iii): OP′ is pending in $\Lambda\gamma'$ or OP′ is initiated in the last $|\Sigma|$ steps of $\Lambda\gamma'\gamma''\Sigma$</u>

   Let p_i be the process that invoked $\text{OP}' = SC(v)$. Consider the most recent *LL* operation from p_i that preceded OP′. Let OP″ denote this operation. We assert that in both Case (ii)

and Case (iii), OP'' completed in Λ. In the next two paragraphs we prove this assertion for the two cases.

In Case (ii), since OP' is pending in $\Lambda\gamma'$, OP'' must have completed in $\Lambda\gamma'$. Since $\text{PSET}(\gamma'\gamma'') = \{p_l\}$ and $p_l \neq p_i$ (because unlike p_i, p_l has no pending operation in $\Lambda\gamma'$), it follows that p_i completed OP'' in Λ.

In Case (iii), since each process appears at most once in Σ, if p_i initiated OP' in the last $|\Sigma|$ steps of $\Lambda\gamma'\gamma''\Sigma$, then it follows that p_i completed OP'' in $\Lambda\gamma'\gamma''$. Further, since $\text{PSET}(\gamma'\gamma'') = \{p_l\}$ and $p_l \neq p_i$ (because $p_l \in \{p_1, \ldots, p_{n-1}\} - \text{PSET}(\Sigma)$), it follows that p_i completed OP'' in Λ.

Since OP did not even begin in Λ, it follows that OP'' is linearized before OP. Thus, we have the following situation: p_i applied the *LL* operation OP'' and then the *SC* operation OP'; p_l's successful *SC* operation OP is linearized after OP'' and before OP'. By the specification of LL/SC bit, OP' must return *false*. This contradicts the premise that OP' is successful. Thus Cases (ii) and (iii) cannot arise. □

3. In $\Lambda\gamma'\gamma''\Sigma\Pi$, if p_n's first operation on $\mathcal{O}$ (which is an *LL* operation) completes, it returns a response different from *res*.

   Proof Since $\text{OP} = SC(w)$ is successful, the value of $\mathcal{O}$ immediately after OP is w. By the previous observation, in $\Lambda\gamma'\gamma''\Sigma$, every successful *SC* linearized after OP is also of the form *SC*(w). Therefore, in $\Lambda\gamma'\gamma''\Sigma\Pi$, p_n's first operation on $\mathcal{O}$ (which is an *LL* operation) returns w. Since $w = 1 - res$, we have $w \neq res$. Hence, we have the observation. □

Lemma 85 is immediate from the last observation. □

6.2.4 Single Writer Snapshot

An *n-process single writer binary snapshot object* [AWW93, And93] supports the operations *read* and *write* v, for $v \in \{0, 1\}$. The states are $[v_1, v_2, \ldots, v_n]$, where $v_1, v_2, \ldots, v_n$ are from $\{0, 1\}$. A *write* x operation from process p_i, when applied in state $[v_1, v_2, \ldots, v_n]$, changes the state to $[v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_n]$ and returns *ack*. The *read* operation, when applied in state $[v_1, v_2, \ldots, v_n]$, returns $[v_1, v_2, \ldots, v_n]$ without affecting the state.

Lemma 86 *Single writer binary snapshot object is perturbable for n processes, for any initial state.*

Proof Consider any linearizable and solo-terminating randomized implementation of an n-process single writer binary snapshot object $\mathcal{O}$, initialized to any value and shared by processes $p_1, \ldots, p_n$. For $1 \leq i \leq n-1$, let *op-list*$_i$ be an infinite sequence of alternating *write 0* and *write 1* operations. Let *op-list*$_n$ be an infinite sequence of *read* operations. Let Λ, Σ, and Π be any schedules that satisfy the four conditions listed in Definition 21.

Recall from (the fourth condition in) Definition 21 that *res* is the value returned by p_n's first operation on $\mathcal{O}$ in $\Lambda\Sigma\Pi$. Let $res = [v_1, v_2, \ldots, v_n]$. Let p_l be any process in $\{p_1, \ldots, p_{n-1}\} - \text{PSET}(\Sigma)$. Let $\gamma \in (\{p_l\} \times \text{COINSPACE})^*$ be the shortest schedule such that, in $\Lambda\gamma$, p_l just completed writing $1 - v_l$. Since the implementation is solo-terminating, γ exists. Further, since

$p_l \in \{p_1, \ldots, p_{n-1}\} - \text{PSET}(\Sigma)$, p_l has no step in the last $|\Sigma|$ steps of $\Lambda\gamma\Sigma$. Therefore, if p_n's first operation in $\Lambda\gamma\Sigma\Pi$ (which is a read operation) completes and returns $[w_1, w_2, \ldots, w_n]$, w_l must equal $1 - v_l$. It follows that $res \neq [w_1, w_2, \ldots, w_n]$. Hence, we have Lemma 86. □

6.3 Applications

In Section 6.2, we showed that every type in set A is perturbable for n processes, for any initial state, where $A = \{$modulo k counter (for any $k \geq 2n$), increment, fetch&add, k-valued compare&swap (for any $k \geq n$), LL/SC bit, single-writer snapshot$\}$ (see Lemmas 81, 82, 83, 84, 85, and 86). From this and Theorem 8, we have:

Theorem 9 *Let A be the set of types defined above. Consider any randomized implementation of an object belonging to a type in A, initialized to any state and shared by processes $p_1, \ldots, p_n$, from resettable consensus objects and historyless objects. If the implementation is linearizable and solo-terminating:*

1. *Its space complexity is at least $n - 1$.*
2. *If the implementation is deterministic, its solo-termination shared-access time complexity is at least $n - 1$.*

The above result does not address the complexity of implementing modulo k counter when $k < 2n$, or of implementing k-valued compare&swap when $k < n$. We discuss these cases below. The following corollary is a simple consequence of Lemma 81.

Corollary 1 *For all $k \geq 1$, modulo k counter is perturbable for $\lfloor k/2 \rfloor$ processes, for any initial state.*

From Corollary 1 and Theorem 8, we have:

Corollary 2 *For any positive integer k, consider any randomized implementation of modulo k counter, initialized to any state and shared by processes $p_1, \ldots, p_{\lfloor k/2 \rfloor}$, from resettable consensus objects and historyless objects. If the implementation is linearizable and solo-terminating:*

1. *Its space complexity is at least $\lfloor k/2 \rfloor - 1$.*
2. *If the implementation is deterministic, its solo-termination shared-access time complexity is at least $\lfloor k/2 \rfloor - 1$.*

We observe that the time or space complexity grows monotonically with the number of processes sharing the implementation. This observation, together with Corollary 2, gives:

Theorem 10 *For any $k \leq 2n$, consider any randomized implementation of modulo k counter, initialized to any state and shared by processes $p_1, \ldots, p_n$, from resettable consensus objects and historyless objects. If the implementation is linearizable and solo-terminating:*

1. *Its space complexity is at least $\lfloor k/2 \rfloor - 1$.*

2. *If the implementation is deterministic, its solo-termination shared-access time complexity is at least* $\lfloor k/2 \rfloor - 1$.

Using Lemma 84 and reasoning as above, we have:

Theorem 11 *For any* $k \leq n$*, consider any randomized implementation of* k*-valued compare&swap, initialized to any state and shared by processes* $p_1, \ldots, p_n$*, from resettable consensus objects and historyless objects. If the implementation is linearizable and solo-terminating:*

1. *Its space complexity is at least* $k - 1$.

2. *If the implementation is deterministic, its solo-termination shared-access time complexity is at least* $k - 1$.

Bibliography

[AD96] H. Attiya and E. Dagan. Universal operations: unary versus binary. In *Proceedings of the 15th Annual Symposium on Principles of Distributed Computing*, pages 223–232, May 1996.

[ADT95] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 538–547, 1995.

[AM95a] J. Anderson and M. Moir. Universal constructions for large objects. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 168–182, 1995.

[AM95b] J. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–194, August 1995.

[AMTT97] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. In *Proceedings of the 16th Annual Symposium on Principles of Distributed Computing*, pages 111–120, August 1997.

[And93] J. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.

[Asp90] J. Aspnes. Time and space efficient randomized consensus. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, 1990.

[Aum97] Y. Aumann. Efficient asynchronous consensus with the weak adversary scheduler. In *Proceedings of the 16th Annual Symposium on Principles of Distributed Computing*, pages 209–218, August 1997.

[AWW93] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the 12th Annual Symposium on Principles of Distributed Computing*, pages 159–170, August 1993.

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

[BL93] J.E. Burns and N.A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107:171–184, 1993.

[Cha96] T. Chandra. Polylog randomized wait-free consensus. In *Proceedings of the 15th Annual Symposium on Principles of Distributed Computing*, pages 166–175, May 1996.

[CJT98] T. D. Chandra, P. Jayanti, and K. Y. Tan. A polylog time wait-free construction for closed objects. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, June/July 1998.

[FHS93] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. In *Proceedings of the 12th Annual Symposium on Principles of Distributed Computing*, pages 241–249, August 1993.

[FHS98] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843–862, September 1998.

[Her88] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, 1988.

[Her91] M.P. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

[Her93] M. Herlihy. A methodology for implementing highly concurrent data structures. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[HS93] M. P. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 111–120, 1993.

[HW90] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[Int02] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 1: Application Architecture Revision 2.1*, October 2002. http://www.intel.com/design/itanium/downloads/245317.htm.

[IR94] A. Israeli and L. Rappoport. Disjoint-Access-Parallel implementations of strong shared-memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 1994.

[Jay98a] P. Jayanti. A lower bound on the local time complexity of universal constructions. In *Proceedings of the 17th Annual Symposium on Principles of Distributed Computing*, June 1998.

[Jay98b] P. Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proceedings of the 17th Annual Symposium on Principles of Distributed Computing*, June 1998.

[JP03] P. Jayanti and S. Petrovic. Efficient and practical constructions of ll/sc variables. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, July 2003.

[JT92] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proceedings of the 6th Workshop on Distributed Algorithms, Haifa, Israel*, November 1992. Appeared in Lecture Notes in Computer Science, Springer-Verlag, No: 647.

[JTT96] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for non-blocking implementations. In *Proceedings of the 15th Annual Symposium on Principles of Distributed Computing*, pages 257–266, May 1996.

[JTT00] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for non-blocking implementations. *SIAM Journal on Computing*, 30(2):438–456, June 2000.

[Knu73] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, second edition, 1973.

[KRS86] C. Kruskal, L. Rudolf, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Proceedings of the 5th Annual Symposium on Principles of Distributed Computing*, pages 218–228, August 1986.

[Lam77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[Moi97a] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, August 1997.

[Moi97b] M. Moir. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, September 1997.

[Plo89] S. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pages 159–175, August 1989.

[Sit92] R. Site. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, 1992.

[ST95] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[Sys02] MIPS Computer Systems. *MIPS64 Architecture for Programmers, Volume II: The MIPS64 Instruction Set*, August 2002. http://www.mips.com/publications/processor_architecture.html.

[TDF+01] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. *IBM eserver POWER4 System Microarchitecture*. IBM, October 2001. http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html.

[TSP92] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock-based concurrent data structure algorithms non-blocking. In *Proceedings of the 11th Symposium on Principles of Database Systems*, pages 212–222, 1992.

[WG] D.L. Weaver and T. Germond. *The SPARC Architecture Manual, Version 9*. SPARC International, Inc. http://soldc.sun.com/articles/sparcv9.pdf.