Generalized Compare-and-Swap and Space-Efficient Universal Constructions for the Infinite-Arrival Model

Vassos Hadzilacos
University of Toronto
Toronto, Canada
vassos@cs.toronto.edu

Myles Thiessen
University of Toronto
Toronto, Canada
mthiessen@cs.toronto.edu

Sam Toueg
University of Toronto
Toronto, Canada
sam@cs.toronto.edu

Abstract

We introduce GCAS, a natural generalization of the well-known compare-and-swap (CAS) object. Intuitively, GCAS just replaces the fixed equality test of CAS with a parametrized comparator chosen from $\{<, =, >\}$. To showcase the utility of GCAS, we present two space-efficient wait-free universal constructions for systems where the number of participating processes is unknown and may be infinite (the *infinite-arrival* model). The first has space-complexity linear in the number of processes that have participated so far, while the second has space-complexity linear in the point contention but assumes bounded concurrency. To the best of our knowledge, these are the first wait-free universal constructions that achieve this space complexity in the infinite-arrival model. To achieve space complexity linear in the point contention, our second universal construction uses a novel memory recycling scheme that works in the infinite-arrival model with bounded concurrency. The ideas behind this recycling scheme could be of more general use.

CCS Concepts

• **Theory of computation → Distributed computing models**; **Concurrent algorithms**; **Algorithm design techniques**.

Keywords

Compare-and-Swap, Universal Constructions, Infinite-Arrival Model

ACM Reference Format:
Vassos Hadzilacos, Myles Thiessen, and Sam Toueg. 2026. Generalized Compare-and-Swap and Space-Efficient Universal Constructions for the Infinite-Arrival Model. In *ACM Symposium on Principles of Distributed Computing (PODC '26), July 6–10, 2026, Egham, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3796701.3815968

1 Introduction

We propose a natural generalization of compare-and-swap (CAS), a fundamental object in shared memory systems, and show how it enables *space-efficient*, wait-free universal constructions in systems where the number of participating processes is unknown and may be infinite (this is the *infinite-arrival model* introduced by Merritt and Taubenfeld [19]). This model encourages the design of *adaptive* algorithms whose performance depends on the number of processes that *actually* participate rather than the maximum number that *could* participate. We now describe our results.

```
CAS(val1, val2)
    current := O
    if current = val1 then
        O := val2
        return TRUE
    return FALSE
```

```
GCAS(C, val1, val2)
    current := O
    if current C val1 then
        O := val2
        return TRUE
    return FALSE
```

Figure 1: CAS and GCAS operations.

A CAS object O supports a CAS(val_1, val_2) operation which compares the current value of O to val_1 and, if equal, replaces it with val_2; see Figure 1 on the left. The object also supports standard read and write operations.

We introduce *generalized compare-and-swap* (*GCAS*), a simple generalization of CAS that replaces the fixed equality test of CAS with a comparator $C \in \{<, =, >\}$ supplied as a parameter. A GCAS object O supports a GCAS(C, val_1, val_2) operation which compares the current value v of O to val_1 using C and, if v C val_1, replaces it with val_2; see Figure 1 on the right. Like CAS objects, GCAS objects also support standard read and write operations. It is worth noting that GCAS should be implementable in hardware with minimal overhead compared to CAS (because testing for inequality is not much harder than testing for equality).[1]

To showcase the utility of GCAS, we present two space-efficient wait-free universal constructions for the infinite-arrival model. The space complexity of our first universal construction at any time t is linear in the number of processes *that have participated by time* t. To the best of our knowledge, this is the first universal construction to achieve this space complexity in the infinite-arrival model.

A drawback of our first construction is that once a process reserves memory, that memory remains allocated forever, even if the process later leaves the system. Ideally, the space complexity at time t would be linear in the number of operations *that are concurrent at time* t, i.e., the point contention at time t. Our second universal construction achieves this space complexity, but under the assumption of an unknown upper bound on the maximum point contention (this is the infinite-arrival model *with bounded*

PODC '26, Egham, United Kingdom

ACM ISBN 979-8-4007-2512-8/26/07
https://doi.org/10.1145/3796701.3815968

[1] In this paper we restrict the comparator C of GCAS to be an equality or inequality test. We do so for two reasons: (a) these tests suffice for our universal constructions, and (b) this restriction minimizes the potential hardware overhead in implementing GCAS relative to CAS. More generally, C could be any other binary comparator such as $\leq$ or $\geq$, or even any function that takes two values and returns TRUE or FALSE.

concurrency [2, 19]). To the best of our knowledge, this is the first universal construction to attain this space complexity in this model.

We achieve this space complexity via a novel memory recycling scheme. Common approaches to memory recycling include *reference counting* (e.g., [3, 9, 14, 22, 23, 25]) and *hazard pointers* or related techniques (e.g., [15, 20, 21]), but, to the best of our knowledge, none of the existing schemes can be used to achieve our goals: some are non-blocking but not wait-free (e.g., [9, 14, 22, 23, 25]), others do not work in the infinite-arrival model (e.g., [3, 7, 15, 20, 21]). Our memory recycling scheme uses reference counters, with a key twist: each reference counter is decomposed into two counters, namely an *acquisitions* and a *revocations* counter, each stored at different locations; these are coalesced back into a single reference counter when its value is needed for recycling.

Our universal constructions leverage GCAS to achieve wait-freedom with a simple helping mechanism that prioritizes operations based on their timestamps. Roughly speaking, to execute an operation o, a process p obtains a timestamp t, and then it competes with other processes to have o selected as the next operation to execute. To do so, p tries to "announce" o by applying a GCAS(>, (t, o), (t, o)) operation on a GCAS "announcement" object A: if the timestamp t of o is smaller than the timestamp t' of the operation o' currently in A, this GCAS operation will replace (t', o') with (t, o) in A (because o has higher priority).[2] Eventually, the operation with the smallest timestamp will "stick" in A and will be executed. Once this operation is executed, however, it must be removed from A *even if it has a higher priority than any current and future operations*. So, if p notices that the timestamped operation (t', o') that is currently in A has been executed, p tries to replace it with its own operation (t, o) by applying a GCAS(=, (t', o'), (t, o)) operation on A.

In summary, this paper makes the following four contributions:

- A natural generalization of the well-known compare-and-swap (CAS) object that replaces the fixed equality test with a parametrized comparator chosen from $\{<, =, >\}$.
- The first wait-free universal construction for the infinite-arrival model whose space complexity at time t is linear in the number of processes that have participated by time t.
- The first wait-free universal construction for the infinite-arrival model with bounded concurrency whose space complexity at time t is linear in the point contention at time t.
- A novel memory recycling scheme for the infinite-arrival model with bounded concurrency. The ideas behind this recycling scheme could be of more general use.

It is worthwhile noting that our first universal construction guarantees more than wait-freedom: the step complexity of each operation is linear in the *point contention* (this is shown in [12]).

Roadmap. In Section 2 we sketch our model. In Section 3 we present a simple universal construction for the infinite-arrival model. In Section 4 we describe our more space-efficient universal construction for the infinite-arrival model with bounded concurrency. In Section 5 we discuss related work. We conclude the paper with some remarks and open problems in Section 6.

[2]Throughout this paper we compare tuples in lexicographic order. In practice, this comparison can be achieved by reserving a field for each component of the tuple, concatenating these fields, and comparing the resulting bit strings.

Due to space limitations, the proofs of our universal constructions are omitted here; they are given in [12].

2 Model Sketch

We consider shared-memory systems where asynchronous processes may fail by crashing. In contrast to most work on shared-memory systems, which assumes a system with n processes (the *n-arrival* model), our system may have an infinite number of processes (the *infinite-arrival* model).

2.1 Objects, Implementations, and Runs

Each object has a type that specifies how the object behaves when it is accessed sequentially. We assume that the type $\mathcal{T}$ of object O is given in the form of a (possibly nondeterministic) state-transition function $apply_{\mathcal{T}}$: if s is a state of O and o is an operation that can be applied to O, $apply_{\mathcal{T}}(o, s)$ returns a pair of the form (s', r), where s' is a possible new state of O and r is the corresponding response returned by o when o is applied to O in state s.

An implementation of a *target* object O from a set of *base* objects is a collection of procedures that specify how any process in the system can perform any operation of O by applying operations to the base objects. We only consider implementations that are *linearizable* [16] and *wait-free* [13]. A universal construction from a set of base objects is an algorithm that takes the state-transition function $apply_{\mathcal{T}}$ of an arbitrary type $\mathcal{T}$ as input, and outputs an implementation of an object of type $\mathcal{T}$ from these base objects.

A run of an implementation of an object O is a sequence of *steps*, where each step is an invocation of an operation on O, an atomic operation on a base object, or a response from an operation on O. Each step in a run R has an associated "time" which is the sequence number of that step within R, i.e., the time of the t-th step in R is t. Furthermore, we say that a process p *has participated* at time t in a run R if p has taken a step in R before or at time t.

2.2 Concurrency

The point contention at time t in a run R is the number of operations that are pending at time t. More precisely:

Definition 1 (Point Contention). *The point contention at time t in a run R is the number of operations that, by time t, have been invoked but have not responded.*

Definition 2 (Bounded Concurrency). *A system has bounded concurrency if for every run R of the system there is a bound $b_R \in \mathbb{N}$ such that the point contention at every time t in R is at most b_R.*

We stress that in a system with bounded concurrency, processes do *not* know the bound on the point contention (so they cannot use it); this is because the bound b_R may be different in each run R.

Recall that our first universal construction works in the infinite-arrival model. This model does *not* assume any bound on concurrency, i.e., there may be runs where the concurrency grows without bound (this is called the infinite-arrival model with *unbounded concurrency* in [2, 19]). Our second universal construction (which is more space efficient than our first) works in the infinite-arrival model with bounded concurrency.

2.3 Memory Manager

To support space efficiency, shared memory systems are often augmented with a memory manager that dynamically allocates and frees cells as instructed by processes. For our purposes, a "cell" consists of a constant number of objects; i.e., it is a structure. The memory manager maintains the set of currently unallocated cells.

When a process p needs a new set of base objects, it asks the memory manager to allocate a new cell comprised of these objects. The memory manager picks a cell C that is not currently allocated, and returns a pointer to C to p. When a process determines that a cell C is no longer needed *by any process*, it asks the memory manager to free it for future reuse; the cell is no longer allocated. We say that a base object is allocated by the memory manager if it resides in a cell that is currently allocated by the memory manager.

We stress that if a process performs an operation on an object in a cell C *that has been freed and not yet reallocated*, the operation may return an incorrect value or may not return at all. This is because after the memory manager regains ownership of C it may use C arbitrarily—for example, it may assign C to another application that accesses it in ways outside the control of our implementation. In particular, this application could modify the contents of C or change its formatting.

In systems with a memory manager, the base objects used by an implementation fall into two categories: *statically allocated objects*, which exist and are known to all processes at the start of a run, and *dynamically allocated objects*, which are currently allocated by the memory manager. So we define the space complexity of an implementation as follows:

Definition 3 (Space Complexity). *The space complexity of an implementation at time t of a run is the number of statically allocated base objects plus the number of base objects dynamically allocated by the memory manager at time t in that run.*

3 A Simple & Space-Efficient Universal Construction

We now describe a simple wait-free universal construction for the infinite-arrival model. Its space complexity at any time is linear in the number of processes that have participated by that time. This universal construction, shown in Algorithm 1, uses GCAS, CAS, and fetch-and-increment (F&I) objects to implement an object O of an arbitrary type $\mathcal{T}$.[3] In this construction, we leverage GCAS to implement the priority-based helping scheme outlined in Section 1.

When a process p invokes its *first* operation, it obtains from the memory manager a pointer to a cell consisting of a single CAS object, and assigns that pointer to a local variable ptr (line 2). Thereafter, across all operations invoked by p, ptr points to this cell. This cell is used to store the response of each operation issued by p; since it is dedicated to p, we will call it *p's cell*. In general it stores a pair (t, r), where t is the timestamp of an operation o on O that p has invoked and r will eventually contain the response of o (initially it is Null, indicating that o is not done yet).

In addition to the cells that store the response of operations, this universal construction uses three statically allocated base objects:

- C (for "clock"): A F&I object used to timestamp operations.
- A (for "announce"): A GCAS object that processes use to announce the operations they wish to apply to O. It contains information about the oldest (highest priority) operation announced that has not yet been applied, namely a tuple (t, o, ptr), where o is an operation, t is its timestamp, and ptr is a pointer to the cell of the process that invoked o.
- S (for "state"): A CAS object that stores information about the state of the target object O. More precisely, it stores a tuple (t, s, r, ptr), where t is the timestamp of the last operation o applied to O, s is the state of O after the application of o, r is the response of o, and ptr is a pointer to the cell of the process that invoked o.

To perform an operation o, a process p first gets a timestamp t for o from the clock object C (line 3). Then, p sets its response cell, which is pointed to by ptr, to (t, Null) (line 4). Operation o may be completed by p itself or by a "helper". While o is not done, i.e., while the response cell of p still contains (t, Null) (line 5):

(1) p reads the tuple (t^*, s^*, r^*, ptr^*) currently in S (line 6).
(2) p ensures that the response of the last operation applied to O is copied into the response cell of the process that invoked it, i.e., p ensures that (t^*, r^*) is written in the cell pointed to by ptr^*, by applying a CAS operation on it (line 7).
(3) p then tries to announce its own operation o by applying a GCAS$(>, \ldots)$ operation on A to write (t, o, ptr) in it. This GCAS will succeed if t is less than the timestamp of the operation presently in A, i.e., if o has higher priority (line 8).
(4) Irrespective of whether this GCAS operation on A was successful (i.e., whether p succeeded in writing (t, o, ptr) in A), p now helps to execute *whatever operation is currently in A*. To do so, p first reads from A the tuple (t', o', ptr') describing the operation to help (line 9), and then it reads the response of o' (in the cell pointed to by ptr') to see whether o' is already done (line 10).
   (a) If o' is not done, p applies o' to the state s^* of the target object O that it read in step (1), to get the new state s' of O and the response r' of o'. It then attempts to linearize o' by trying to replace the tuple it read from S in step (1) with (t', s', r', ptr') (lines 11-13).
   (b) If o' is done, p tries to remove o' from A by replacing it with its own operation o. To do so, p applies a GCAS$(=, \ldots)$ operation on A to replace (t', o', ptr') with (t, o, ptr) (line 15).

When p finds that o is done, it returns the response of o, which is stored in the response cell of p (line 16).

Theorem 4 ([12]). *Algorithm 1 is a wait-free universal construction for the infinite-arrival model. Its space complexity at time t is linear in the number of processes that have participated by time t.*

In fact, this universal construction guarantees more than wait-freedom: we prove that the step complexity of each operation is linear in the *point contention* (at the time the invoking process gets a timestamp for this operation). More precisely:

Theorem 5 ([12]). *Suppose a process p invokes an operation o and executes line 3 within o. Let c be the point contention at this time. Then, the number of steps that p takes within o is at most linear in c.*

[3] Since GCAS is a generalization of CAS, we can replace all CAS objects with GCAS objects. But, to highlight where the additional functionality of GCAS is used, we opted to use CAS rather than GCAS whenever CAS is sufficient.

Statically allocated shared objects:

C : A F&I object, initially 1.

A : A GCAS object with three fields:
- *time* : the timestamp t of the operation o in the field below, initially 0.
- *operation* : the operation o to execute, initially Noop.
- *pointer* : a pointer ptr to the cell of the process that invoked o, initially a pointer ptr_{Noop} to a dummy cell.

S : A CAS object with four fields:
- *time* : the timestamp t of the operation o that resulted in the current object state s, initially 0.
- *state* : the current object state s, initially the initial state of type $\mathcal{T}$.
- *response* : the response r of the operation o that resulted in the current object state s, initially $\perp$.
- *pointer* : a pointer ptr to the cell of the process that invoked o, initially ptr_{Noop}.

Dynamically allocated shared objects:

Each cell consists of a single CAS object with two fields:
- *time* : the timestamp t of the last operation o invoked by the owner of this cell.
- *response* : the response r of the operation o or Null.

Local persistent variable per process:

ptr : a pointer to the cell of this process, initially Null.

```
1  procedure DoOp(o)                                          // to perform an operation o on the target object
2      if ptr = Null then ptr := AllocateCell()               // get a pointer to a cell from the memory manager
3      t := F&I(C)                                                                  // get a timestamp t for o
4      (*ptr) := (t, Null)                                                  // initialize o's response to Null
5      while (*ptr) = (t, Null) do                          // while o is not done (i.e., its response is Null)
6          (t*, s*, r*, ptr*) := S                                          // read the state of the target object
7          CAS((*ptr*), (t*, Null), (t*, r*))       // copy the response of the last operation into the appropriate cell
8          GCAS(>, A, (t, o, ptr), (t, o, ptr))                          // if o has higher priority, announce it
9          (t', o', ptr') := A                          // read the currently announced operation o' to help
10         (t̂, r̂) := (*ptr')                            // read the response of o' from the appropriate cell
11         if (t̂, r̂) = (t', Null) then                    // if o' is not done (i.e., its response is Null)
12             (s', r') := apply_T(o', s*)                                  // apply o' to the target object state
13             CAS(S, (t*, s*, r*, ptr*), (t', s', r', ptr'))                // try to linearize o' by changing S
14         else                                                // if o' is done (i.e., its response is not Null)
15             GCAS(=, A, (t', o', ptr'), (t, o, ptr))                  // and o' is still in A, try to announce o
16     return (*ptr).response                                  // return the response of o (found in o's cell)
```

Algorithm 1: A simple & space-efficient wait-free universal construction.

4 A More Space-Efficient Universal Construction

The space complexity of our first universal construction at time t is linear in the number of processes that have participated by t. We now describe a universal construction whose space complexity at t is linear *only* in the point contention at t (Definition 1). It uses the same types of base objects as our first universal construction, except it also uses fetch-and-add (F&A) in addition to fetch-and-increment (F&I). Both constructions work in the infinite-arrival model, but the second one requires the additional assumption of bounded concurrency (Definition 2). We first outline the main challenges in achieving this space complexity and how our universal construction solves them, and then present its pseudocode.

4.1 Some Challenges and Their Solutions

In our first universal construction, the first time a process participates, it gets a new response cell from the memory manager and never frees it. In other words, this construction never recycles these cells. To improve the space complexity, our second universal construction recycles cells, i.e., it *frees* previously allocated cells.

Cell recycling. One difficulty with recycling is that before a process p frees a cell C, it must be sure that no process will ever try to access an object O within C until C is allocated again. This is because if this were to happen, O could misbehave: it could return a wrong value, or even not return at all.

A naive way to recycle cells with our first universal construction is as follows. When a process p invokes an operation o, it allocates a new cell C to store the response of o. Then, after p finds the response of o in C (on line 16), it immediately frees C (because p no longer needs it). The problem with this approach is that another process q can now access C, even though C is unallocated. This occurs when q reads a pointer to C from the announce object A on line 9 (before C has been freed), goes to sleep, wakes up after C has been freed, and then accesses C on line 10.

A common approach to enable the freeing of no-longer-needed cells is by using *reference counters* (e.g., [3, 9, 14, 22, 23, 25]). Intuitively, a reference counter for a cell C stores the number of processes that currently have the right to access C. A process acquires the right to access C by incrementing the reference counter for C; and when it no longer needs to access C, the process decrements the reference counter for C. A process that finds a cell's reference counter to be 0 can free that cell.

But where do we put the reference counter for a cell C? If we put it in C itself, then to acquire the right to access C (by incrementing its reference counter) a process would have to access C (where its reference counter is stored) — a chicken-and-egg situation. To solve this, we could try to put C's reference counter outside of C. But doing this raises another problem: when we recycle C, we now must also recycle its reference counter; so we need a mechanism to recycle the reference counters themselves — a different kind of chicken-and-egg situation!

Our second universal construction solves this problem by (a) threading the cells of the operations that are pending in a *linked list*, and (b) *splitting* the reference counter of each cell into two parts, each in a different location, as we now explain.

Reference counter splitting. At any time t, the reference counter for a response cell C in the list is equal to the number of processes that have *acquired* the right to access C *minus* the number of processes that have *relinquished* that right by time t. We store the reference counter for C *implicitly* by maintaining two separate counters: the *acquisitions counter* for C, stored in the *predecessor of C in the list*; and the *revocations counter* for C, stored *in C itself*. Note that the acquisitions counter for C and the pointer to C, both of which reside in the predecessor of C, must be updated *together* atomically. So we store both of them in a CAS object called *next* (in the predecessor of C). The revocations counter for C is stored in a F&A object, called *revocations*, in C itself.

List traversal. To access any cell C in the list, a process p must first acquire the right to do so, i.e., it must increment the acquisitions counter for C. Since this counter is located in the predecessor of C, p must *traverse* the list to find (and acquire the right to access) the predecessor of C. This traversal proceeds as follows. Having obtained the right to access a cell C_i in the list, p first obtains the right to access the next cell C_{i+1} by incrementing the acquisitions counter for C_{i+1}, which resides in C_i (p does so by performing a successful CAS operation on the *next* object of C_i because it contains the acquisitions counter for C_{i+1}). After p has acquired the right to access C_{i+1}, it no longer needs access to C_i, so it relinquishes its right to access C_i. It does so by incrementing C_i's revocations counter, by performing a F&A operation on the *revocations* object of C_i. We note that the starting point of this traversal, i.e., the head of the list $H = C_0$, is a *statically* allocated cell that is never freed (so all processes always have the right to access H).

Cell removal. When an operation completes, its corresponding response cell is removed from the list. To remove a cell C from the list, a process p must move the acquisitions counter for the *successor* C^+ of C, stored in C, to the *predecessor* C^- of C. But the removal of C and the move of the acquisitions counter for C^+ (from C to C^-) must be done atomically to avoid the following bad scenario. Before p removes C from the list, it reads the acquisitions counter for C^+ from C, say its value is a. Then, another process acquires the right to access C^+ by incrementing the acquisitions counter for C^+ stored in C; at this time, the number of acquisitions for C^+ is $a+1$. Now p removes C from the list and writes a into C^-. But the acquisitions counter for C^+, now stored in C^-, is incorrect: its value is a, but the true number of acquisitions for C^+ is $a+1$! To solve this problem, p removes C from the list as follows: (1) it first *freezes* the acquisitions counter for C^+, (2) it then reads the acquisitions counter for C^+, say its value is a, and (3) it finally removes C from the list and writes a into C^-; this last step is done atomically by doing a CAS operation on the *next* object of C^-. Process p does step (1) by setting a *sealed* flag in the *next* object of C (which contains the acquisitions counter for C^+). Once this flag is set, the content of *next* cannot change (the content of the *next* object of C is now *sealed*).

Cell freeing. When a cell C is removed from the list it cannot necessarily be freed yet. This is because some process may still have the right to access C. To determine when C can be freed, we need to determine whether the reference counter for C is zero. This is done as follows. When a process p removes C from the list, it computes the reference counter for C by: (1) reading the value a of the acquisitions counter for C, which is stored in the predecessor of C, and (2) subtracting a from the revocations counter for C, which is stored in C, using a F&A operation on the *revocations* object of C. Note that this subtraction changes the semantics of the *revocations* object of C: it used to be the revocations counter for C, it is now the *negation* of the reference counter for C. This trick allows processes to relinquish their right to access C in a *uniform* way by incrementing the *revocations* object of C (irrespective of its current semantics). In [12], we prove that the process that causes the reference counter for C to become zero is the last process that had the right to access C, so it can safely free C.

Wait-freedom. To achieve wait-freedom, we use a modified version of the priority-based helping mechanism of Algorithm 1. But this is no longer sufficient here, because processes that are trying to use the list (e.g., traverse the list, add a cell, remove a cell, or change the content of a cell) may be prevented from doing so by other processes that are concurrently using the list. So we also need a mechanism to ensure that accessing the list is wait-free. We now briefly elaborate on these two mechanisms.

To apply an operation o on the target object O, a process p first obtains from the memory manager a cell C to store the response of o. Then p performs the following three "low-level" operations, possibly with the help of other processes, in that order:

(1) AddCell: append C to the end of the list;
(2) Apply&CopyResponse: apply o to O and then copy the response to C; and
(3) RemoveCell: remove C from the list.

A process performs these three operations using a modified version of the priority-based helping mechanism of Algorithm 1. As before, process p first obtains a timestamp for the operation that it wants to do from a F&I object C, it tries to announce it by applying a GCAS(>, ...) operation on the announce object A, and then tries to perform the operation o_A stored in A. Recall that in Algorithm 1, p tries to perform o_A as follows: (a) it first reads the state s of O from the state object S, (b) it then applies o_A to s (using the state-transition function $apply_{\mathcal{T}}$ of the type $\mathcal{T}$ of O) to get the

next state s' of O and the corresponding response r', and (c) it finally tries to write (s', r') into the state object S by doing a CAS operation on that object. Note that, if this CAS is successful, step (3) does two things simultaneously: it linearizes o_A *and* changes the state of O accordingly. In contrast, our second universal construction separates these two things, as follows.

When p tries to perform the operation o_A stored in A, it simply tries to write o_A into a CAS object L. We ensure that once an operation is written into L, *it is not removed until it has taken effect.* Thus, operations are linearized in the order they are written into L (which is why this object is called L).

We now explain how processes perform the operations written into L. Recall that in Algorithm 1, before doing its own operation, a process p *copies the response* of the last operation that was linearized (into the appropriate cell). In contrast, in our second universal construction, before doing its own operation, a process p *performs* the last operation that was linearized. To do so, p reads the operation o_L that is currently in L, and then:

(1) If o_L is an AddCell operation to add a cell C, p traverses the list to append C to the end of the list.
(2) If o_L is an Apply&CopyResponse operation to apply an operation o to the target object O, p first applies o to O by performing steps (a), (b) and (c) above to the state object S. Then p traverses the list to find the appropriate cell and copies the response of o into it.
(3) If o_L is a RemoveCell operation to remove a cell C, p traverses the list to find and remove C from the list.

The priority-based helping mechanism described above, however, is not sufficient for performing the operation o_L that is currently in L in a wait-free manner: as we see above, to perform o_L, a process p must traverse the list of cells, but this traversal could be impeded by concurrent processes that are also traversing the list. To see this, recall that to reach the successor C^+ of a cell C in the list, p must acquire the right to access C^+ (by incrementing the acquisitions counter for C^+, which resides in C). But to do so p must "win" a CAS operation on the *next* object of C which contains the acquisitions counter for C^+. This is problematic because p may keep losing its CAS operations on the *next* object of C, so p *may get stuck at cell* C while trying to traverse the list. To avoid this, in our universal construction, p periodically checks whether the operation o_L that it is trying to perform is still in L (recall that p is traversing the list to perform o_L). If p sees that o_L is no longer in L, it can be certain that o_L has already taken effect (because o_L cannot be removed from L until it has taken effect), and so p *bails out.*

This bail-out mechanism to achieve wait-freedom, however, works only under the assumption of bounded concurrency. This is because, with *un*bounded concurrency, a process attempting to traverse the list may repeatedly lose its CAS operations on a *next* object because there may be an unbounded stream of newly arriving processes, each of which wins a CAS operation on that *next* object and then immediately crashes *before changing* L.

Different incarnations of a cell. Recycling cells may also raise the following problem. A process p reads a *next* object that contains a pointer to a cell C, but goes to sleep before acquiring the right to access C (i.e., before incrementing the acquisitions counter for C, which also resides in this *next* object). Then C gets recycled and reallocated; this is a new "incarnation" of C, and its content has changed. Finally, p wakes up and acquires the right to access (the new incarnation of) C, thinking that it has acquired the right to access the older incarnation of C — this is clearly problematic. A simple solution to this problem is for processes to tag each pointer returned by the memory manager with a *unique identifier* (which they can obtain by performing a F&I operation): this creates "unique pointers" that are used in place of "raw" pointers everywhere (except for when a process needs a "raw" pointer to access a cell). As it turns out, creating these unique pointers is not necessary: in [12] we show that the above scenario (and other problematic ones involving different incarnations of a cell) cannot occur in our universal construction.

Space complexity. In [12], we show that at any time t, our universal construction uses a number of cells that is linear in the point contention c_t at time t. Intuitively, this follows from the following properties:

- At any time, a process holds the right to access *at most a small constant number* α *of cells.* This is ensured by relinquishing access to each cell as soon as it is no longer needed (for example, during list traversal, a process successively acquires and relinquishes cells as it traverses through the list).
- When a process relinquishes the right to access a cell, it decrements the cell's (implicit or explicit) reference counter. If the resulting value indicates that no process currently holds the right to access the cell, it frees the cell and returns it to the memory manager.
- Before completing an operation on the target object, a process relinquishes the right to access every cell it acquired the right to access during that operation.

The above properties ensure that the universal construction uses about $\alpha \cdot c_t$ cells (so about $3\alpha \cdot c_t$ base objects) at any time t.

4.2 Pseudocode Description

The pseudocode of this universal construction is given in Algorithm 2. This algorithm uses the following statically allocated base objects: H (for "head"), C (for "clock"), A (for "announce"), L (for "linearize"), and S (for "state"). This algorithm also uses a list of dynamically allocated cells (H is the head of this list). Each cell consists of three base objects: *response*, *revocations*, and *next*. The purpose of all the base objects was described in the previous section, and their type and content is given at the top of Algorithm 2. Note that *next* is a CAS object with four fields: *view*, *sealed*, *acquisitions*, and *ptr*. We already described the purpose of *sealed*, *acquisitions*, and *ptr* in the previous section. The *view* field is a monotonically increasing counter that prevents ABA problems.[4]

• **DoHighLevelOp(***hlo***)** is invoked by any process p that wants to perform an operation *hlo* on the target object (to differentiate *hlo* from the three low-level operations that our universal construction does to perform *hlo*, we call *hlo* a "high-level" operation). In line 2 p gets a pointer *ptr* to a new cell from the memory manager. Process p then performs the operation *hlo* on the target object by ensuring the following three low-level operations are performed in order: (1) add the cell pointed to by *ptr* to the list of cells, (2) apply *hlo* to the target object and copy its response into the cell pointed to by *ptr*,

[4]This field can be avoided by making *next* an LL/SC object rather than a CAS object.

Statically allocated shared objects:

H : A cell that serves as the head of a list of cells.

C : A F&I object, initially 1.

A : A GCAS object with two fields:
- *ullo* : a unique low-level operation to linearize, initially $(0, \text{Noop})$.
- *ptr* : a pointer to a cell in which to store the response of *ullo*, initially Null.

L : A CAS object with two fields:
- *ullo* : the last unique low-level operation that was linearized, initially $(0, \text{Noop})$.
- *ptr* : a pointer to a cell in which to store the response of *ullo*, initially Null.

S : A CAS object with three fields:
- *ullo* : the unique Apply&CopyResponse operation that resulted in the current object state, initially $(0, \text{Noop})$.
- *state* : the current object state, initially the initial state of type $\mathcal{T}$.
- *resp* : the response of *ullo*, initially $\perp$.

Dynamically allocated shared objects:

The response of each high-level operation is stored in a cell; a cell consists of the following objects

type cell:
- *response* : A CAS object with two fields:
  - *ullo* : the last unique low-level operation invoked by the owner of this cell, initially $(0, \text{Noop})$.
  - *resp* : the response of applying this low-level operation or Null, initially Null.
- *revocations* : A F&A object that stores the number of times this cell was revoked, initially 0.
- *next* : A CAS object with four fields:
  - *view* : the number of times *next* has changed in this incarnation of this cell, initially 0.
  - *sealed* : True if this cell is in the process of being removed otherwise False, initially False.
  - *acquisitions* : the number of times the cell pointed to by *ptr* was acquired, initially 0.
  - *ptr* : a pointer to a cell, initially Null.

```
1  procedure DoHighLevelOp(hlo)
2      ptr := AllocateCell()
3      DoLowLevelOp(AddCell, ptr)
4      DoLowLevelOp(⟨Apply&CopyResponse, hlo⟩, ptr)
5      (−, resp) := (*ptr).response
6      DoLowLevelOp(RemoveCell, ptr)
7      Relinquish(ptr)
8      return resp
9  procedure DoLowLevelOp(llo, ptr)
10     t := F&I(C)
11     ullo := (t, llo)
12     (*ptr).response := (ullo, Null)
13     while (*ptr).response = (ullo, Null) do
14         (ullo_L, ptr_L) := L
15         if ullo_L = (*, AddCell) then
16             DoAddCell(ullo_L, ptr_L)
17         else if ullo_L = (*, RemoveCell) then
18             DoRemoveCell(ullo_L, ptr_L)
19         else if ullo_L = (*, ⟨Apply&CopyResponse, *⟩) then
20             DoApply&CopyResponse(ullo_L, ptr_L)
21         GCAS(>, A, (ullo, ptr), (ullo, ptr))
22         (ullo_A, ptr_A) := A
23         status := IsDone(ullo_L, ullo_A, ptr_A)
24         if status = NotDone then
25             CAS(L, (ullo_L, ptr_L), (ullo_A, ptr_A))
26         else if status = Done then
27             GCAS(=, A, (ullo_A, ptr_A), (ullo, ptr))
28 procedure DoAddCell(ullo_L, ptr_L)
29     curr_ptr := &H
30     while curr_ptr ≠ ptr_L do
31         (status, next_ptr) := AcquireNext(ullo_L, curr_ptr)
32         if status = L-Changed then break
33         else if status = NotFound then
34             (view, −, −, −) := (*curr_ptr).next
35             if L.ullo ≠ ullo_L then break
36             CAS((*curr_ptr).next, (view, False, 0, Null), (view + 1, False, 1, ptr_L))
37             break
38         else if status = Found then
39             Relinquish(curr_ptr)
40             curr_ptr := next_ptr
41     Relinquish(curr_ptr)
42     SetResponse(ullo_L, ptr_L, Done)
43 procedure DoRemoveCell(ullo_L, ptr_L)
44     SetResponse(ullo_L, ptr_L, Done)
45     (prev_ptr, curr_ptr) := (Null, &H)
46     while curr_ptr ≠ ptr_L do
47         (status, next_ptr) := AcquireNext(ullo_L, curr_ptr)
48         if status = L-Changed ∨ status = NotFound then
49             goto line 63
50         else if status = Found then
51             Relinquish(prev_ptr)
52             (prev_ptr, curr_ptr) := (curr_ptr, next_ptr)
53     while (*ptr_L).next.sealed = False do
54         (view, sealed, a, next_ptr) := (*ptr_L).next
55         CAS((*ptr_L).next, (view, sealed, a, next_ptr), (view + 1, True, a, next_ptr))
56     (−, −, a, next_ptr) := (*ptr_L).next
57     repeat
58         (view′, −, a′, next_ptr′) := (*prev_ptr).next
59         if L.ullo ≠ ullo_L ∨ next_ptr′ = next_ptr then
60             goto line 63
61     until CAS((*prev_ptr).next, (view′, False, a′, ptr_L), (view′ + 1, False, a, next_ptr))
62     F&A((*ptr_L).revocations, −a′)
63     Relinquish(prev_ptr)
64     Relinquish(curr_ptr)
65 procedure DoApply&CopyResponse(ullo_L, ptr_L)
66     (ullo_S, s_S, r_S) := S
67     if L.ullo ≠ ullo_L then return
68     if ullo_S ≠ ullo_L then
69         (−, ⟨Apply&CopyResponse, hlo_L⟩) := ullo_L
70         (s, r) := apply_T(hlo_L, s_S)
71         CAS(S, (ullo_S, s_S, r_S), (ullo_L, s, r))
72         r_S := S.resp
73     SetResponse(ullo_L, ptr_L, r_S)
74 procedure SetResponse(ullo_L, ptr_L, response)
75     status := Acquire(ullo_L, ptr_L)
76     if status = Found then
77         CAS((*ptr_L).response, (ullo_L, Null), (ullo_L, response))
78         Relinquish(ptr_L)
79 procedure IsDone(ullo_L, ullo_A, ptr_A)
80     status := Acquire(ullo_L, ptr_A)
81     if status = L-Changed then return L-Changed
82     response := Done
83     if ullo_A = (*, AddCell) ∧ status = NotFound then
84         response := NotDone
85     else if ullo_A = (*, RemoveCell) ∧ status = Found then
86         response := NotDone
87     else if ullo_A = (*, ⟨Apply&CopyResponse, *⟩) then
88         if (*ptr_A).response = (ullo_A, Null) then
89             response := NotDone
90     if status = Found then Relinquish(ptr_A)
91     return response
92 procedure Acquire(ullo_L, target_ptr)
93     curr_ptr := &H
94     while curr_ptr ≠ target_ptr do
95         (status, next_ptr) := AcquireNext(ullo_L, curr_ptr)
96         Relinquish(curr_ptr)
97         if status = L-Changed ∨ status = NotFound then
98             return status
99         else if status = Found then
100            curr_ptr := next_ptr
101    return Found
102 procedure AcquireNext(ullo_L, curr_ptr)
103    repeat
104        (view, −, a, next_ptr) := (*curr_ptr).next
105        if L.ullo ≠ ullo_L then
106            return (L-Changed, *)
107        if next_ptr = Null then
108            return (NotFound, *)
109    until CAS((*curr_ptr).next, (view, False, a, next_ptr), (view + 1, False, a + 1, next_ptr))
110    return (Found, next_ptr)
111 procedure Relinquish(curr_ptr)
112    if curr_ptr = Null ∨ curr_ptr = &H then return
113    x := F&A((*curr_ptr).revocations, 1)
114    if x + 1 = 0 then FreeCell(curr_ptr)
```

Algorithm 2: A more space-efficient wait-free universal construction.

and (3) remove this cell from the list. This is done by invoking the DoLowLevelOp procedure with a first parameter of AddCell, ⟨Apply&CopyResponse, *hlo*⟩, and RemoveCell on lines 3, 4, and 6, respectively. After p has finished these procedures, *hlo* is done. Hence, p no longer needs the cell pointed to by *ptr* (which was used to store the response of *hlo*), so it relinquishes its right to access it (line 7). Recall that the cell pointed to by *ptr* is not necessarily recyclable yet because other processes may still have the right to access it. Finally, p returns the response of *hlo* (line 8) that it read from the cell pointed to by *ptr* on line 5.

• **DoLowLevelOp(***llo*, *ptr***)** performs the low-level operation *llo* and ensures the response of this operation is stored in the cell pointed to by *ptr*. The high-level flow of this procedure is similar to Algorithm 1. Process p first gets a unique timestamp t (line 10) and forms the pair (t, llo); we call the pair (t, llo) a *unique low-level operation* and denote it *ullo* (line 11). The timestamp t is the priority of *llo*. Process p then sets the response of *ullo* in the cell pointed to by *ptr* to Null (line 12), and enters the loop on line 13. p exits this loop once the response of *ullo* is not Null (indicating *ullo* has taken effect).

In each iteration of this loop: (1) p reads the value $(ullo_L, ptr_L)$ currently stored in L, and then invokes the DoAddCell, DoRemoveCell, or DoApply&CopyResponse procedure, depending on the kind of operation $ullo_L$ is, to ensure $ullo_L$ takes effect and its corresponding response is written into the cell pointed to by ptr_L (lines 15-20); (2) p tries to announce its *own* operation and pointer (*ullo*, *ptr*) by performing a GCAS(>, . . .) operation on A (line 21); (3) regardless of the outcome of this GCAS operation, p reads the low-level operation $ullo_A$ currently stored in A (line 22), and then it invokes the IsDone procedure to try to determine whether this operation has already taken effect (line 23). If IsDone returns NotDone, then $ullo_A$ has not yet taken effect and p tries to store $(ullo_A, ptr_A)$ into L by performing a CAS operation on L (line 25). If IsDone returns Done, then $ullo_A$ has taken effect, and in this case p tries to remove $(ullo_A, ptr_A)$ from A by replacing it with $(ullo, ptr)$ by performing a GCAS(=, . . .) operation on A (line 27). Finally, if IsDone returns L-Changed, then p could not determine whether $ullo_A$ has taken effect or not; in this case, p does nothing: it just proceeds to the next iteration of the loop.

• **DoAddCell(**$ullo_L$**,** ptr_L**)** traverses the list of cells to find the last cell in the list, appends the cell C pointed to by ptr_L after it, and sets its response to Done. To traverse the list, p maintains a current pointer *curr_ptr* that points to some cell in the list (*curr_ptr* is initially a pointer to the head of the list $\&H$). While *curr_ptr* does not point to C (i.e., $curr\_ptr \neq ptr_L$), p tries to acquire the right to access the cell after the cell pointed to by *curr_ptr* by invoking the AcquireNext procedure (line 31). If the AcquireNext procedure returns Found (line 38), then p continues the traversal: p relinquishes its right to *curr_ptr* (line 39), updates *curr_ptr* to the next pointer (line 40), and continues to the next iteration of the while loop. If the AcquireNext procedure returns NotFound (line 33), *curr_ptr* points to the last cell C_{last} in the list. Therefore, p did not find C in the list, and p tries to add C after C_{last}. To do so, p performs a CAS operation on C_{last} to set its next pointer to ptr_L (line 36). In [12], we prove that, regardless of whether this CAS succeeds or not, C is added to the list at the time of this CAS. So, in all cases, p exits the while loop after performing this CAS. Note that the CAS that adds C to the list sets the acquisitions counter for C to be 1 to signify that the process that allocated C has the right to access C (this process will relinquish its right to access C at the end of its operation on line 7). After p exits the while loop, p first relinquishes its right to access the cell pointed to by *curr_ptr* (line 41). Then, p invokes the SetResponse procedure to write Done into the *response* object of C; this informs the process that allocated C that C has been added to the list (line 42).

• **SetResponse(**$ullo_L$**,** ptr_L**,** *response***)** tries to set the response of $ullo_L$ in the cell C pointed to by ptr_L to *response*. To do so, p tries to acquire the right to access C by invoking the Acquire procedure on line 75; if it is successful, then p tries to write *response* into the *response* object of C by performing a CAS operation on this object (line 77). In [12], we prove that by the time p exits this procedure, the *response* object of C was set to *response* (this is true even if p was unsuccessful in acquiring the right to access C or p failed the CAS on line 77). Note that, before exiting this procedure, if p was successful in acquiring the right to access C, then p relinquishes its right to access C on line 78.

• **AcquireNext(**$ullo_L$**,** *curr_ptr***)** assumes that the process p invoking it has the right to access the cell C pointed to by *curr_ptr*, and it returns one of the following: (Found, *next_ptr*), meaning p has acquired the right to access the next cell in the list (i.e., the one pointed to by *next_ptr*); (NotFound, ∗), meaning there is no cell after the one pointed to by *curr_ptr* in the list (so C is the last cell of the list); and (L-Changed, ∗), meaning the operation stored in L is no longer $ullo_L$ (which implies that $ullo_L$ has already taken effect, so there is no need to acquire the right to access the cell after C). To acquire the right to access the next cell C^+ in the list, p enters a repeat-until loop (line 103) in which it repeatedly tries to increment the acquisitions counter for C^+, which is stored in C. To do so, p first reads the *next* object of C to learn the current number of acquisitions a of C^+ and the pointer *next_ptr* to C^+ (line 104). Then, p performs a CAS operation on the *next* object of C to set the acquisitions counter for C^+ to $a + 1$ (line 109). If this CAS operation succeeds, p has acquired the right to access C^+. There are three ways that p can exit this repeat-until loop: (1) the CAS operation on line 109 succeeds, in which case p returns (Found, *next_ptr*); (2) p finds that C is the last cell in the list, so C^+ does not exist, in which case p returns (NotFound, ∗); or (3) p finds that L no longer stores $ullo_L$, in which case p returns (L-Changed, ∗).[5]

• **Acquire(**$ullo_L$**,** *target_ptr***)** traverses the list of cells to find the cell pointed to by *target_ptr* and acquires the right to access it. This procedure returns one of the following: Found, meaning p has acquired the right to access the cell pointed to by *target_ptr*; NotFound, meaning the cell pointed to by *target_ptr* is not in the list; and L-Changed, meaning the operation stored in L is no longer $ullo_L$. To determine whether the cell pointed to by *target_ptr* is in the list, p searches for this cell by traversing the list, starting from the head H of the list. To do so, p maintains a pointer *curr_ptr* to the current cell C that it has reached in its traversal. Process p initializes *curr_ptr* to $\&H$, and then it enters the while loop on line 94 which continues until $curr\_ptr = target\_ptr$, i.e., until p finds the cell pointed to by *target_ptr*; at this point p has acquired the right

[5]This is the "bail out" mechanism described in Section 4.1. Recall that this is needed because p could be stuck trying to acquire C^+ because its CAS operations fail forever.

to access it. In each iteration of this loop, p invokes the AcquireNext procedure to try to acquire the right to access the successor C^+ of the cell C pointed to by *curr_ptr*. Then, irrespective of the result, it relinquishes the right to access C. If p fails in acquiring the right to access C^+, then p returns L-CHANGED or NOTFOUND (depending on the reason why it failed). Otherwise, (i.e., if p succeeds in acquiring the right to C^+) p sets *curr_ptr* to the pointer of C^+. If C^+ is the cell pointed to by *target_ptr*, then p will exit the loop and return FOUND.

• **IsDone(**$ullo_L$**,** $ullo_A$**,** ptr_A**)** checks whether $ullo_A$ has taken effect. This procedure returns one of the following: NOTDONE, meaning $ullo_A$ has not taken effect; DONE, meaning $ullo_A$ has taken effect; and L-CHANGED, meaning the operation stored in L is no longer $ullo_L$. Observe that this procedure *cannot* simply check if $ullo_A$ has already taken effect by just reading the *response* object of the cell C_A pointed to by ptr_A. This is because p *cannot* access any cell (including C_A) before acquiring the right to access it, i.e., incrementing the acquisitions counter for C_A (which is located in the predecessor of C_A in the list). So, p first tries to acquire the right to access C_A by executing the Acquire procedure on line 80; let *status* be its response. If *status* = L-CHANGED, then IsDone returns L-CHANGED. Otherwise, *status* equals NOTFOUND or FOUND. At this point, p can determine whether $ullo_A$ has taken effect or not as follows: $ullo_A$ has *not* taken effect if and only if $ullo_A$ is an ADDCELL operation and *status* = NOTFOUND (which means that the Acquire procedure did not find C_A in the list); $ullo_A$ is a REMOVECELL operation and *status* = FOUND (which means that the Acquire procedure found C_A in the list); or $ullo_A$ is an APPLY&COPYRESPONSE operation, and the response in C_A is still ($ullo_A$, NULL). If p determines that $ullo_A$ has not taken effect, the IsDone procedure returns NOTDONE, and otherwise returns DONE. Before returning, however, p relinquishes its right to access C_A if the Acquire procedure found C_A in the list, i.e., if *status* = FOUND.

• **Relinquish(***curr_ptr***)** assumes that the process p invoking it has the right to access the cell C pointed to by *curr_ptr*, and it is used by p to relinquish its rights to access C. To do so, p increments the *revocations* object of C by doing a fetch-and-increment operation on it. Let c be the value of this object immediately after this fetch-and-increment operation. As we explained in Section 4.1, when $c > 0$, c is the value of the *revocations* counter for C; otherwise, $|c|$ is the value of the *reference* counter for C. In [12] we prove that if $c = 0$, this relinquish by p is the last relinquish for C, and so p can safely free C.[6]

• **DoApply&CopyResponse(**$ullo_L$**,** ptr_L**)** is used by a process p to: (1) apply the high-level operation hlo_L stored in $ullo_L$ (where $ullo_L = (*, \langle$APPLY&COPYRESPONSE$, hlo_L\rangle)$) to the target object state stored in S, and (2) copy its response into the cell C pointed to by ptr_L. To do so, p first reads the content $(ullo_S, s_S, r_S)$ of S (line 66). Recall that $ullo_S = (*, \langle$APPLY&COPYRESPONSE$, hlo_S\rangle)$ where hlo_S is the last high-level operation to the target object that has taken effect, s_S is the current state of the target object, and r_S is the response of hlo_S. Then, p checks if $ullo_L$ is still stored in L (line 67); if L has changed, then hlo_L has already been applied to the target object and the response of hlo_L has already been copied into C, so p exits the procedure. Otherwise, $ullo_L$ is still stored in L. In this case, in [12], we prove that hlo_L has been applied to the target object if and only if $ullo_S = ullo_L$. So p now checks whether $ullo_S \neq ullo_L$ (line 68), and if so p tries to apply hlo_L to the target object. To do so, p determines the new state s and response r by applying hlo_L to s_S using the state-transition function $apply_{\mathcal{T}}$ of the type $\mathcal{T}$ of the target object (line 70), and tries to write $(ullo_L, s, r)$ into S using a CAS operation (line 71). In [12], we prove that regardless of whether this CAS is successful or not, hlo_L has been applied to the target object. What remains to be done is to copy the response from S into C. To do so, p first reads the current response from S (line 72) (p must re-read the response from S because of non-determinism: the response that p got from $apply_{\mathcal{T}}$ may be different from the response that the process that succeeded in writing the response into S received). Then p invokes the SetResponse procedure to copy this response into the cell C pointed at by ptr_L.

• **DoRemoveCell(**$ullo_L$**,** ptr_L**)** is used by a process p to remove the cell C pointed to by ptr_L from the list if p finds C in the list. Process p does this in four stages: (1) p traverses the list of cells to find C and its predecessor C^- (lines 45-52); (2) if p finds C, p seals the acquisitions counter stored in C to prevent any process from acquiring the right to access the successor C^+ of C (lines 53-55); (3) then it removes C from the list by setting the successor of C^- to C^+ (lines 56-61); and (4) finally p consolidates the acquisitions and revocations counters for C into the reference counter for C (line 62).

To do stage (1), p traverses the list of cells by maintaining a current pointer *curr_ptr* and a pointer to its predecessor *prev_ptr*, which are initially $\&H$ and NULL, respectively. While *curr_ptr* does not point to C (i.e., $curr\_ptr \neq ptr_L$), p tries to acquire the right to access the cell after the cell pointed to by *curr_ptr* by invoking the AcquireNext procedure (line 47). If the AcquireNext procedure returns FOUND (line 50), then p continues the traversal: p relinquishes its right to *prev_ptr* (line 51), updates (*prev_ptr*, *curr_ptr*) to *curr_ptr* and the next pointer, respectively (line 52), and continues to the next iteration of the while loop. If the AcquireNext procedure returns L-CHANGED or NOTFOUND (line 48), then p skips stages (2), (3), and (4): C has already been removed from the list.

To do stage (2), while the *next* object of C is not sealed, p repeatedly performs a CAS operation on the *next* object of C to try to set its *sealed* flag from FALSE to TRUE (line 55). In [12], we prove that, even though these CAS operations can keep failing, the *next* object of C is eventually sealed, and so p eventually exits this loop.

To do stage (3), p repeatedly performs a CAS operation on the *next* object of C^- on line 61 to try to: (a) copy the acquisitions counter for C^+ (which is in C) into C^-; and (b) set the successor of C^- to C^+. There are two ways p can exit this loop. First, if p finds that C has already been removed from the list (either because the successor of C^- is C^+ or the operation $ullo_L$ to remove C from the list is no longer stored in L), then p immediately exits this loop. Second, if p performs a successful CAS operation on line 61, then p is the process that removes C from the list.

To do stage (4), p must now compute the reference counter for C from the acquisitions and revocations counters for C (which are located in C^- and C, respectively), and store it in C. To do so, p decrements the revocations counter for C by the value of the acquisitions counter that p previously read from C^- (on line 58). Let v be the value of the *revocations* object of C after this decrement.

[6] In the pseudocode, $c = x + 1$ because fetch-and-increment fetches the value of the object *before* incrementing it.

Note that (a) $|v|$ now represents the *reference* counter for C; and (b) $v < 0$ (because p still has the right to access C).

At this point (just after line 62), C has been removed from the list (and its reference counter was computed). The natural thing to do now would be to set the *response* object of C to Done (by invoking SetResponse($ullo_L, ptr_L$, Done) between line 62 and 63), *but this does not work* (we give a scenario to illustrate why in [12]). So, we set the *response* object of C to Done by invoking SetResponse($ullo_L, ptr_L$, Done) in the *first line* of the DoRemoveCell procedure. In other words, we tell the process that wants to remove C from the list that C has been removed from the list *before we actually remove C from the list!* Although this does not look right, in [12], we prove that it does not affect the universal construction's correctness or asymptotic space complexity. Roughly speaking, this is because the operation to remove C, namely $ullo_L$, is in L, and so no other operation can now occur unless this removal is done first.

Theorem 6 ([12]). *Algorithm 2 is a wait-free universal construction for the infinite-arrival model with bounded concurrency. Its space complexity at time t is linear in the point contention at t.*

5 Related Work

Most object implementations in shared-memory systems do not work in the infinite-arrival model: they assume a system with n processes, where n is known to the processes (this is the *n-arrival model* [2]). Object implementations for these systems typically use this known n in their code, and they use some number of base objects that depends on this n. These base objects are statically allocated in *every* run, even those in which fewer than n processes actually participate. This is clearly undesirable.

To avoid this, researchers have designed algorithms for systems where the number of processes is bounded but unknown (this is called the *finite-arrival model* [2]), e.g., algorithms in [2, 17, 21]. But such algorithms may not work in the infinite-arrival model, i.e., if an infinite number of processes may participate in a run. For example, as Aguilera pointed out in [2], the simple naming algorithm in [2, Figure 4] is not wait-free in these runs.

Researchers have also designed algorithms for the infinite-arrival model (e.g., [1, 2, 4–6, 10, 18, 19, 24]). Existing algorithms for this model are typically not space-efficient, and some use infinitely many objects in every run (e.g., [1, 2, 4, 10, 19]). In particular, the universal construction of [4] uses infinitely many base objects in every run because it relies on the collect algorithm of [10], and the universal construction of [24] uses space linear in the number of *operations* applied so far. Observe that this can be much higher than the number of *processes* that have participated so far, because each participating process can apply arbitrarily many operations. So, the space complexity of [24] can be much higher (and never less) than the space complexity of our first universal construction.

Other work has focused on designing space-efficient algorithms for the infinite-arrival model. For example, the LL/SC implementation from CAS in [18] uses a number of base objects linear in the number of processes that have participated so far.

The universal constructions of [5, 6] were also developed for the infinite-arrival model and aim to achieve space efficiency. However, the constructions of [5, 6] rely on an external garbage collection mechanism that *automatically frees* any object that "becomes inaccessible by any process in the system" [5], even though no such mechanism is provided. Determining when an object becomes inaccessible and can be safely reclaimed is itself a difficult problem [7]. Indeed, to the best of our knowledge, there is no known automatic wait-free garbage collection mechanism for the infinite-arrival model. In contrast, our universal constructions do not assume any garbage collection mechanism. In particular, in our second algorithm, processes explicitly manage the recycling of objects.

It is worth noting that the algorithms given in [5, 6] are analyzed under a space complexity measure that accounts only for the space used *at quiescent times*, i.e., only at times when no operations are executing. But this measure provides no bounds on the space used in runs without quiescent times, i.e., runs in which at every moment at least one operation is executing.

We note that the infinite-arrival model and its variants, including versions with bounded and unbounded concurrency, were introduced by Merritt and Taubenfeld in their seminal paper [19]. The GCAS object and our first universal construction originally appeared in [11]. This universal construction was inspired by the *2-nonblocking* universal construction of [8]; in particular, as in our construction, processes compete on a single announce object.

6 Conclusion

We introduced GCAS, a simple and natural generalization of CAS, and showed how it can be used to obtain two space-efficient, wait-free universal constructions in the infinite-arrival model. The first has space-complexity linear in the number of processes that have participated so far, the second has space-complexity linear in the point contention but assumes bounded concurrency.

A natural question is whether such universal constructions can be achieved using CAS instead of GCAS. Equivalently, can GCAS be implemented in a space-efficient manner using CAS in the infinite-arrival model?

If the answer is yes, then plugging such an implementation into our algorithms would immediately yield space-efficient, wait-free universal constructions based on CAS. If the answer is no, this would demonstrate that GCAS is strictly more powerful than CAS for at least one purpose: obtaining space-efficient, wait-free universal constructions in the infinite-arrival model.

We conclude with a final open question: can one achieve the space complexity of our second universal construction in the infinite-arrival model *without assuming bounded concurrency?*

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This work was partially supported by the NSERC Discovery award RGPIN-2022-03304. The second author was supported by an Ontario Graduate Scholarship.

References

[1] Yehuda Afek, Eli Gafni, and Adam Morrison. 2006. Common2 extended to stacks and unbounded concurrency. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*. 218–227.
[2] Marcos K Aguilera. 2004. A pleasant stroll through the land of infinitely many creatures. *ACM Sigact News* 35, 2 (2004), 36–59.
[3] Daniel Anderson, Guy E Blelloch, and Yuanhao Wei. 2021. Concurrent deferred reference counting with constant-time overhead. In *Proceedings of the 42nd ACM*

SIGPLAN International Conference on Programming Language Design and Implementation. 526–541.

[4] James Aspnes, Gauri Shah, and Jatin Shah. 2002. Wait-free consensus with infinite arrivals. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 524–533.

[5] Denis Bédin, François Lépine, Achour Mostéfaoui, Damien Perez, and Matthieu Perrin. 2021. Wait-Free CAS-Based Algorithms: The Burden of the Past. In *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[6] Denis Bédin, François Lépine, Achour Mostéfaoui, Damien Perez, and Matthieu Perrin. 2024. Wait-free Algorithms: the Burden of the Past. (2024). https://doi.org/10.21203/rs.3.rs-4125819/v1

[7] Trevor Alexander Brown. 2015. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. 261–270.

[8] David YC Chan, Shucheng Chi, Vassos Hadzilacos, and Sam Toueg. 2021. Differentiated nonblocking: a new progress condition and a matching queue algorithm. *arXiv preprint arXiv:2103.11926* (2021).

[9] David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele Jr. 2001. Lock-free reference counting. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. 190–199.

[10] Eli Gafni, Michael Merritt, and Gadi Taubenfeld. 2001. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. 161–169.

[11] Vassos Hadzilacos, Myles Thiessen, and Sam Toueg. 2024. Generalized Compare and Swap. *arXiv preprint arXiv:2410.19102* (2024).

[12] Vassos Hadzilacos, Myles Thiessen, and Sam Toueg. 2026. Generalized Compare-and-Swap and Space-Efficient Universal Constructions for the Infinite-Arrival Model. *arXiv preprint arXiv:2605.19237* (2026). Full version of this paper.

[13] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149.

[14] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. 2005. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)* 23, 2 (2005), 146–196.

[15] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2002. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *International Symposium on Distributed Computing*. Springer, 339–353.

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

[17] Prasad Jayanti and Srdjan Petrovic. 2005. Efficiently implementing a large number of LL/SC objects. In *International Conference On Principles Of Distributed Systems*. Springer, 17–31.

[18] Prasad Jayanti and Srdjan Petrovic. 2005. Efficiently implementing LL/SC objects shared by an unknown number of processes. In *International Workshop on Distributed Computing*. Springer, 45–56.

[19] Michael Merritt and Gadi Taubenfeld. 2013. Computing with infinitely many processes. *Information and Computation* 233 (2013), 12–31.

[20] Maged M Michael. 2002. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*. 21–30.

[21] Maged M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504.

[22] Ruslan Nikolaev and Binoy Ravindran. 2019. Hyaline: fast and transparent lock-free memory reclamation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 419–421.

[23] Ruslan Nikolaev and Binoy Ravindran. 2021. Snapshot-free, transparent, and robust memory reclamation for lock-free data structures. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 987–1002.

[24] Matthieu Perrin, Achour Mostefaoui, and Grégoire Bonin. 2020. Extending the wait-free hierarchy to multi-threaded systems. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*. 21–30.

[25] John D Valois. 1995. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. 214–222.