Common2 Extended to Stacks and Unbounded Concurrency

Yehuda Afek
School of Computer Science
Tel-Aviv University
afek@tau.ac.il

Eli Gafni
Computer Science Department
University of California, Los Angeles
eli@cs.ucla.edu

Adam Morrison
School of Computer Science
Tel-Aviv University
adamx@tau.ac.il

ABSTRACT

Common2, the family of objects that implement and are wait-free implementable from 2 consensus objects, is extended inhere in two ways: First, the `stack` object is added to the family – an object that was conjectured not to be in the family. Second, Common2 is investigated in the unbounded concurrency model, whereas until now it was considered only in an n-process model.

We show that `fetch-and-add`, `test-and-set`, and `stack` are in Common2 even with respect to this stronger notion of wait-free implementation. This necessitated the wait-free implementation of immediate snapshots in the unbounded concurrency model, which was previously not known to be possible.

In addition to extending Common2, the introduction of unbounded-concurrency may help in resolving the Common2 membership problem: If, as conjectured, `queue` is not implementable for a-priori known concurrency n, then it is definitely not implementable for unbounded concurrency. Proving the latter should be easier than proving the former. In addition we conjecture that the `swap` object, that has an n-process implementation, does not have an unbounded concurrency implementation.

Categories and Subject Descriptors

F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*; C.2 [**Computer-Communication Networks**]: Miscellaneous; B.3.2 [**Memory Structures**]: Design Styles—*shared memory*

General Terms

Algorithms, Theory

Keywords

Common2, Wait-free, Stack, Queue, Consensus number 2, Unbounded concurrency, Immediate snapshot

PODC'06, July 22-26, 2006, Denver, Colorado, USA.

1. INTRODUCTION AND RELATED WORK

Understanding the conditions under which a *wait-free* implementation of an object may exist is a fundamental issue in distributed computing. In [11] Herlihy characterized objects by their *consensus number* — the maximum number of processes that can reach agreement using copies of that object and read/write registers. Herlihy showed that an n-consensus object is universal for n processes, i.e., any object can be implemented by n processes in a system with n-consensus objects, without causing the indefinite postponement of any single process by other processes that may be either faster or slower.

This raises the question whether every consensus power c object is wait-free implementable from c-consensus objects in a system with $n > c$ processes. For the case $c = 1$, Herlihy exhibits a nondeterministic object of consensus power 1 that does not have a wait-free read/write implementation for $2 > 1$ processes [10]. The $c = 2$ case was addressed by Afek, Weisberger and Weisman in [3], where they define Common2, the class of objects that have wait-free implementation for any $n > 2$ processes using 2-consensus objects. They show that the commonly used primitives `test-and-set` (`TAS`), `fetch-and-add` (`F&A`), and `swap` are in Common2.

The status of the `queue` and `stack` objects, which also have consensus power 2, remained open. Afek et. al. tried and failed to provide a wait-free implementation of a queue from 2 consensus objects, and it was then conjectured that queue has no such implementation.

Many attempts have been made to decide whether `queue` and `stack` are in Common2. There is an implementation of a *partially defined* queue from Common2 objects, in which a dequeue performed on an empty queue is undefined and will never complete [13]. Li constructed a linearizable queue in a restricted concurrency setting, where only two processes may perform dequeue operations [14]. David gave a queue implementation for any number of dequeuers but only supporting a single enqueuer [5]. David conjectured that Common2 objects cannot even implement a three process queue.

Continuing this line of work, David, Brodsky and Fich gave Common2 implementations of a queue in which only one value can be enqueued and of a stack supporting any number of poppers but at most two pushing processes [6]. They conjectured that a stack supporting three poppers, three pushers and more than a single data value is impossible to implement using only Common2 objects.

In this paper a simple wait-free `stack` implementation

from 2-consensus objects is provided, thus showing that stack is in Common2 and refuting the above conjecture. Thereafter, we extend the study of Common2 to the *unbounded concurrency* model of [8, 15].

In the standard n-process model [11] in which n processes arrive repeatedly, the maximum concurrency at any point is n, and it is hereafter called the *n-bounded* concurrency model. In [8, 15] a system with an infinite set of processes is assumed and the concurrency may increase without bound as more and more processes join the execution. An additional model considered in this paper is the *finite* concurrency model in which the concurrency is finite but unbounded, i.e., there is no a-priori bound n on the concurrency that may be encountered during the execution although the concurrency is known to be finite.

The Common2 implementations in [2, 3] depend on n-bounded concurrency and do not work in the finite concurrency or unbounded concurrency model. In this paper we provide unbounded concurrency implementations of `F&A` and `TAS` from 2-consensus objects. These are then used to turn the n-bounded `stack` implementation into an unbounded concurrency implementation.

The main construction which is the corner stone for the others, is the `F&A`. While we follow the general approach of [2] in constructing `F&A`, we had to substantially modify it to work in the unbounded concurrency model. In [2], first the one-shot immediate snapshot of Borowsky and Gafni [4] is used to build a long-lived immediate snapshot. This long-lived immediate snapshot is then used to construct an atomic write-and-snapshot object which is basically the same as a `F&A`. All of the above implementations from [2, 4] are in the n-bounded concurrency model.

Our first step is to devise an unbounded concurrency immediate snapshot, a construction left open in [8]. This implementation has the same structure as the long-lived immediate snapshot implementation in [2] but it requires a finite concurrency immediate snapshot building block. Luckily such a building block has been recently provided in [7].

The `F&A` implementation in [2] modifies their long-lived immediate snapshot implementation rather than treat it as a black box. Our approach is more modular and uses the immediate snapshot as a black box. To do this, we use the idea from [7] of processes "shepherding tokens" on behalf of other processes in the `F&A` implementation. Our implementation of the "shepherding tokens" idea is much simpler than the protocol used in [7], since in our context we can use `TAS`.

Finally, we spent quite some time in an attempt to construct a `swap` object in the unbounded concurrency model, and we conjecture that `swap` is not wait-free implementable in that model. Thus, `swap` appears to be an object which is wait-free implementable in the n-bounded model but not in the unbounded concurrency model. In contrast, the `queue` object is conjectured not to be implementable in either model.

The paper is organized as follows: Section 2 describes our standard terms and terminology. In Section 3 we describe the stack implementation in the n-bounded model and describe how to turn it into an unbounded concurrency implementation. In Section 4, we describe the construction of our new building blocks of unbounded concurrency immediate snapshot and `F&A`. We conclude in Section 5 with a summary of our results. Proofs are relegated to the appendix.

2. MODEL AND DEFINITIONS

Our model of an asynchronous shared memory system follows [13]. The system consists of processes that communicate by applying operations to shared *base objects*. Our base objects are standard read/write registers, `swap`, `test-and-set` (`TAS`) and `fetch-and-add` (`F&A`).

An *implementation* of a high-level object O is an algorithm specifying the base object operations that each process needs to perform in order to return a response when a invoking a high-level operation on O. We consider only wait-free implementations.

We investigate several concurrency levels. In the *n-bounded concurrency* model there are at most n processes active in any execution. In the *finite concurrency* model, every execution contains a finite number of processes, but there is no fixed bound on this number. The *unbounded concurrency* model allows for an infinite number of processes to be active in an execution, and the concurrency may grow without bound as the execution progresses.

A *stack* is an object supporting the operations Push() and Pop(). Its *state* is a sequence of values $[x_m, \ldots, x_0 = \perp]$ where x_m is called the *top* of the stack. The state $[\perp]$ is the *empty* state. A Push(y) operation in state $[x_m, \ldots, x_0 = \perp]$ changes the state to $[y, x_m, \ldots, x_0 = \perp]$. A Pop() operation in state $[x_m, \ldots, x_0 = \perp]$ returns the top of the stack, x_m, and if $x_m \neq \perp$, the Pop operation changes the state to $[x_{m-1}, \ldots, x_0 = \perp]$.

3. STACK IMPLEMENTATION

In this section we describe a wait-free stack implementation from read/write registers and Common2 objects. The implementation is based on the partial queue implementation of [11].

We initially describe (Algorithm 1) a stack implementation from `swap` and `F&A` objects. The implementation itself is unbounded concurrency except that the `swap` building block does not have an unbounded concurrency implementation. We then provide an alternative implementation (Algorithm 2) without a `swap`, that uses `TAS` and `F&A`, two objects for which unbounded concurrency implementations are given in the next section.

The reason for presenting the `swap`-based stack (Algorithm 1) first is that it highlights the main issues of the implementation and provides the best insight to understanding the linearizability proof. The later usage of `TAS` (Algorithm 2) is mainly a technicality.

The `swap`-based stack implementation given in Algorithm 1 uses an infinite array `items` of `swap` registers initialized to `NULL`, and a `F&A` register named `range`. To push a value, a process obtains an `items` cell index by incrementing `range` and then writes the value to that cell. A popping process reads `range` by adding 0 to it, and then goes down the `items` array from `range` to the first cell, performing a `swap` operation on each cell until a non-`NULL` value is returned from a `swap`. The process then returns that value. Otherwise, the process returns $\perp$ (empty) after reaching the bottom of the array without successfully `swap`ping a non-`NULL` value.

For the exposition only, we assume w.l.o.g. that the values pushed onto the stack are unique, and denote a Pop operation that returns value v by Pop($\uparrow v$), and a Pop operation that returns $\perp$ by Pop($\uparrow\perp$). We refer to cells in the `items` array by their indexes. The index to which a Push(x)

Algorithm 1 Wait-free `stack` implementation using `swap`

```
   Shared variables:
       range : F&A object initialized to 1
       items : array [1, ...] of swap object initialized to NULL

   procedure Push(x)
1:     i := F&A(range, 1)
2:     items[i] := x
   end Push

   procedure Pop()
3:     t := F&A(range, 0) - 1
       for i := t downto 1
4:         x := swap(items[i], NULL)
5:         if x ≠ NULL then return x
       end for
6:     return ⊥
   end Pop
```

Algorithm 2 Wait-free `stack` using `TAS` instead of `swap`

```
   Shared variables:
       range : F&A object initialized to 1
       items : array [1, ...] registers
       T : array [1, ...] of TAS objects

   procedure Push(x)
1:     i := F&A(range, 1)
2:     items[i] := x
   end Push

   procedure Pop()
3:     t := F&A(range, 0) - 1
       for i := t downto 1
4:         x := items[i]
5:         if x ≠ NULL then
6:             if TAS(T[i]) then return x
           end if
       end for
7:     return ⊥
   end Pop
```

operation writes value x is denoted $index(x)$, and we define $index(\perp) = 0$.

The linearizability proof is presented in Appendix A. It relies on the following observations. If Pop($\uparrow$ x) is active concurrently with Push(x), they can be linearized next to each other at any point of time where both are active without affecting the state of the stack (this was also observed in [9]). In the proof, we start off with a history H of a stack execution and eliminate all such pairs from it, leaving us with a new history H'.

The proof then shows that in an execution like H', with no concurrent Push(x)/Pop($\uparrow$ x) operations, there is a point t during each Pop($\uparrow$ v)'s execution such that for every cell c, $c \geq index(v)$, that is non-NULL at t, Pop($\uparrow$ $val^t(c)$) is active at t, where $val^t(c)$ denotes the (non-NULL) value stored in cell c at time t. Thus, the operations Pop($\uparrow$ $val^t(c)$) for every non-NULL cell c, $c \geq index(v)$, can all be linearized at t, one after the other according to the order of the values in `items`.

To linearize a Push(x) operation, we consider the time t_x at which Push(x) writes to `items` in H'. It could be that for every cell c, $c \geq index(x)$, that is non-NULL at t_x, Pop($\uparrow$ $val^{t_x}(c)$) has already been linearized. Thus in the linearized stack x is at the top of the stack. In this case, we linearize Push(x) at t_x. Otherwise, there is some value y whose Pop has not yet been linearized at t_x, with $index(y) > index(x)$. For every such y, Push(y) performs its `F&A` after Push(x) performs its `F&A`. We therefore linearize Push(x) immediately before Push(y'), where y' is such a y with lowest index. Since Pop($\uparrow$ y') has not been linearized by t_x, all the high-level operations linearized after Push(y') remain legal according to the stack specification.

Since Algorithm 1 is clearly wait-free, we thus have:

THEOREM 1. *Algorithm 1 is a wait-free n-bounded* `stack` *implementation from read/write registers and n-bounded* `swap` *and* `F&A` *objects.*

Since `swap` and `F&A` have n-bounded implementations from 2-consensus objects [3], we obtain the following corollary.

COROLLARY 2. *Algorithm 1 is a wait-free n-bounded* `stack` *implementation from read/write registers and 2-consensus objects.*

In order to turn Algorithm 1 into an unbounded concurrency implementation, we use the unbounded concurrency `F&A` and `TAS` implementations from Section 4. Using `swap` allowed a Pop operation to obtain its return value while being assured that it is the only operation to return that value. To use `TAS` instead of `swap`, we have the Pop operations check that a value has actually been written to `items` before trying to capture it using `TAS`.

This is shown in Algorithm 2, where a `TAS` object `T[`i`]` is associated with each cell i in `items`. Instead of performing a `swap`, a process reads `items[`i`]` (Line 4) to verify that a value has been placed in that cell. If the value read is non-`NULL`, the process tries to capture that value by performing a `TAS` on `T[`i`]` (Lines 5-6).

Note that any execution of Algorithm 2 can be reduced into an execution of Algorithm 1, as follows: (1) replace a `read` event that returns `NULL` with a `swap` that returns `NULL` (Line 4); (2) remove `read` events that return non-`NULL`; (3) replace a winning `TAS` at `T[`i`]` with a `swap` that returns `items[`i`]`; and (4) replace a losing `TAS` at `T[`i`]` with a `swap` that returns `NULL`. Thus, the same linearizability arguments used to prove Algorithm 1 correct apply to Algorithm 2. Using the unbounded concurrency implementations of `F&A` and `TAS` from 2-consensus objects provided in the next section, we thus obtain the following:

THEOREM 3. *Algorithm 2 is a wait-free unbounded concurrency* `stack` *implementation from read/write registers and unbounded concurrency* `TAS` *and* `F&A` *objects.*

COROLLARY 4. *Algorithm 2 is a wait-free unbounded concurrency* `stack` *implementation from read/write registers and 2-consensus objects.*

4. EXTENDING COMMON2 TO UNBOUNDED CONCURRENCY

In this section we follow and modify the approach of [2] to develop an unbounded concurrency `F&A` implementation.

We provide a new *atomic write-and-snapshot* implementation, the key building block used to implement `F&A` in [2]. This new atomic write-and-snapshot algorithm uses ideas from [7] and is of interest by itself.

The n-bounded `F&A` construction of [2] utilizes snapshot and immediate snapshot algorithms. A snapshot algorithm provides to each participating process P a set S_P of processes that invoked the algorithm such that, for any processes P and Q: (1) $P \in S_P$, (2) $S_P \subseteq S_Q$ or $S_Q \subseteq S_P$, and (3) if P terminates before Q starts then $Q \notin S_P$. An *immediate snapshot* (IS) [4] has an additional property: that $Q \in S_P$ implies $S_Q \subseteq S_P$.

The n-bounded `F&A` algorithm of [2] is based on the one-shot n-bounded immediate snapshot algorithm of [4]. Both use a technique of processes descending through *levels*.

In the one-shot n-bounded immediate snapshot, a process starts by announcing (in a SWMR register) that it is active at Level n. The process then counts the number of processes active at levels $\leq n$. If there are n processes active at levels $\leq n$, the process stops and returns these processes as its immediate snapshot. Otherwise, it descends to level $n-1$ and repeats. This continues until the process arrives at some level L where it sees L processes active in levels $\leq L$. These processes are then returned as the process' immediate snapshot.

In [2], the one-shot n-bounded immediate snapshot is used as a building block to derive a long-lived n-bounded immediate snapshot algorithm. This algorithm is then modified to obtain an n-bounded `F&A` algorithm. `F&A` is obtained from the long-lived immediate snapshot by associating a `TAS` object with each level, so that a process may return a snapshot from level L only if it wins the `TAS` of Level L. Each process thus obtains a unique snapshot, and summing the inputs of all the processes in that snapshot yields a `F&A` implementation.

We devise an unbounded concurrency immediate snapshot implementation in Section 4.1, and use this immediate snapshot to implement `F&A` in Section 4.2. The novelty in our `F&A` implementation is that, unlike the n-bounded `F&A` of [2], we use immediate snapshot as a black box and do not modify the immediate snapshot algorithm. This is enabled by using ideas from [7] of processes "shepherding tokens" on behalf of other processes.

4.1 Unbounded concurrency immediate snapshot

Here we describe an implementation of immediate snapshot (IS) from read/write registers for the unbounded concurrency model.

For ease of presentation we describe a one-shot implementation in which each process performs at most one immediate snapshot. This implementation can easily be turned into a long-lived one by having each process P simulate a sequence $P_1, P_2, \ldots$ of different processes in the one-shot implementation, where P_i performs the i-th operations of P.

The immediate snapshot object supports a single `ImmSnap` operation, which writes the invoking process' name and returns a snapshot. The unbounded concurrency `ImmSnap` implementation is presented in Algorithm 3.

Like the one-shot n-bounded immediate snapshot of [4], Algorithm 3 is based on the idea of processes descending through *levels*, with a process that *stops* at Level L returning the processes that it sees active at levels $\leq L$ as its immediate snapshot. But while in the n-bounded algorithm processes always start their descent at Level n, in the unbounded concurrency setting a process first needs to determine the level at which it start its descent.

Algorithm 3 Unbounded concurrency immediate snapshot (Code of process P)

```
   Shared variables:
       Level : array [0, ...] of sets, initially empty
       Flag : two-dimensional array [1, ...] × [1, ...] of
              registers, initially FALSE
       IS : array [0, ...] of finite concurrency immediate
            snapshot objects

   procedure ImmSnap()
1:     S := Snapshot()
2:     Level[|S|] := S                    // Post my snapshot
3:     MyLevel := min j such that P ∈ Level[j] // j ≤ |S|
4:     for L := MyLevel − 1 downto 0
5:         if Level[L] ≠ ∅ then Flag[L][P] := TRUE
6:         CurIS := IS[L].ImmSnap()
       // IS[L] is a finite concurrency immediate snapshot
7:         if exists Q ∈ CurIS with Flag[L][Q] = TRUE then
8:             if P ∉ Level[L] return Level[L] ∪ CurIS
9:         end if
10:    end for
11:    return Level[0] ∪ CurIS // Return the immediate
                               // snapshot of Level 0
   end ImmSnap
```

To do this, when a process P invokes `ImmSnap` it first takes a snapshot S of all participating processes. The snapshot is taken using the unbounded concurrency snapshot algorithm of [8] (Line 1). With each snapshot size L we associate a level register `Level[`L`]`. Upon obtaining a snapshot S at level (size) $|S|$, P posts S to `Level[`$|S|$`]` (Line 2).

Next, P scans the `Level` array to find the smallest sized snapshot in which P appears (Line 3). Such a snapshot exists at `Level[`$|S|$`]`, so this is done in a finite number of steps. Say the obtained snapshot's size is MyLevel. P then starts its descent at Level MyLevel $-$ 1. This ensures that any process that arrives after P (and hence obtains a snapshot containing P) will start its descent at a level above P's MyLevel.

P now descends as far down as it can, until it finds a Level L whose associated snapshot in `Level[`L`]` does not contain P. The process Q that obtained and posted the snapshot at Level L may have already returned an immediate snapshot that does not contain P. In fact, Q's `ImmSnap` invocation may have terminated before P started executing. Thus, P cannot descend any further and must return the processes active at levels $\leq L$ as its immediate snapshot.

But how does P find out the processes active at levels $\leq L$? To do this, these processes are partitioned into two sets: processes that started their descent at a level below L, and the processes that descend through Level L. The properties of the snapshot object imply that the snapshot posted to `Level[`L`]` contains the processes that start at a level below L.

To determine the processes that enter Level L, we associate a finite concurrency immediate snapshot object with

each level. This immediate snapshot is implemented using the algorithm of [7], whose key property is that it works for any *finite* number of processes and does not require a-priori knowledge of the number of processes that access it. Later on, in the correctness proof, we show that each such immediate snapshot is indeed accessed by a finite number of processes.

When entering a level L, every process first participates in the immediate snapshot associated with Level L. In doing this, the process simultaneously obtains the list of all processes that have entered the level before it (or concurrently to it), and leaves its own mark for the processes that arrive later.

Thus, when P decides to stop at Level L, the immediate snapshot it returns is the union of the level's snapshot (read from `Level[`L`]`) together with the immediate snapshot obtained upon entering Level L.

The above procedure takes place in Lines 4-11. P starts descending through the levels MyLevel $-\ 1, \ldots, 0$ (Line 4). At each level L, P checks if there is a snapshot posted at Level L. If P sees a snapshot posted to Level L, it reports this fact by setting a flag in a register associated with P for that level (Line 5).

P now participates in the immediate snapshot associated with level L (Line 6). If the immediate snapshot P obtains contains a process Q whose flag for level L is set, P checks whether the snapshot of Level L contains itself. If not, it is possible that another process has already stopped at Level L and returned an immediate snapshot that does not contain P. Hence P cannot descend any further and must also return its immediate snapshot from Level L. P therefore returns the union of Level L's snapshot with `CurIS`, the immediate snapshot that P obtained at Level L (Lines 7-8).

Otherwise, P continues down in the same manner. If P falls all the way down without successfully obtaining a snapshot then P exits the loop with $L = 0$. In this case P obtains an immediate snapshot at Level 0 as usual and terminates the loop, since a snapshot of size 0 is never posted. P then returns the immediate snapshot obtained from Level 0 (Line 11).

We now turn to prove the implementation's correctness:

THEOREM 5. *Algorithm 3 is a wait-free unbounded concurrency immediate snapshot implementation from read/write registers.*

First, we show that the algorithm is indeed wait-free. This follows from the following lemma.

LEMMA 6. *The immediate snapshot object at each level is accessed a finite number of times.*

PROOF. Suppose the claim is false. Then there is an execution of Algorithm 3 in which the immediate snapshot object IS[L] is accessed an infinite number of times. Hence there is some time t at which IS[L] is accessed by a process P that obtained a snapshot of size $L' > L$ in Line 1. Prior to descending into Level L, P writes its snapshot to Level L'. Therefore, any process that obtains a snapshot of size $> L'$ after time t will not descend below Level L'. Hence only a finite number of processes can descend past Level L' after time t, which is a contradiction. □

COROLLARY 7. *Algorithm 3 is wait-free.*

To conclude the proof of Theorem 5, we show that the snapshots returned by Algorithm 3 possess all the immediate snapshot properties.

LEMMA 8. *Algorithm 3 correctly implements an immediate snapshot.*

PROOF. We use the following notation. L_P is the level that process P returns its snapshot from, and $IS_P = \text{Level}[L_P] \cup \text{CurIS}_P$ is the set that P returns.

For any processes P and Q, if $L_P = L_Q$ then the immediate snapshot properties hold for CurIS_P and CurIS_Q, so they hold for IS_P and IS_Q.

Otherwise, w.l.o.g. $L_P > L_Q$. Thus $\text{Level}[L_Q] \subset \text{Level}[L_P]$. For any $R \in IS_Q \setminus \text{Level}[L_Q]$, if $R \notin \text{Level}[L_P]$ then the minimum size snapshot containing R must be above Level L_P. Hence when R moved through Level L_P, all the processes in the immediate snapshot R obtained from `IS[`L_P`]` had their flag set to `FALSE`. Since CurIS_P contains a process with its flag set to `TRUE`, $R \in \text{CurIS}_P$. Thus, $IS_Q \subseteq IS_P$. Clearly $P \notin IS_Q$ and so we have the immediate snapshot properties. □

4.2 Construction of F&A and TAS

Here we construct an unbounded concurrency `F&A` using registers and instantiations of the n-bounded `TAS` object of [3] for infinitely many values of n. Given `F&A` it is trivial to implement `fetch-and-inc` and `TAS`. The `TAS` implementation can be made resetable using the techniques of [1].

As in [2], our `F&A` implementation is based on an *atomic write-and-snapshot*. This is a stronger form of an immediate snapshot, one which guarantees that *each* process obtains a *unique* snapshot. Given an atomic write-and-snapshot, implementing `F&A` is simple: a process P announces its input and obtains an atomic write-and-snapshot S_P, returning the sum of all inputs announced in S_P. The code is given in Algorithm 4.

Algorithm 4 Unbounded concurrency fetch and add

```
   Shared variables:
       WS : atomic write and snapshot object
       Args : array [1, ...] of registers, initially empty

   procedure F&A(k)
1:     Args[P] := k
2:     S := WS.WriteAndSnap()
3:     return ∑_{Q∈S, Q≠P} Args[Q]
   end F&A
```

Since two processes can reach consensus using an atomic write-and-snapshot, read/write registers alone do not suffice to implement it. In [2], an n-bounded write-and-snapshot is constructed by first building a long-lived n-bounded immediate snapshot algorithm, and then modifying that algorithm to use n-bounded `TAS` to ensure that each process returns a unique snapshot. Our approach is different and more modular: we use immediate snapshot in a black box manner, without changing the internals of the immediate snapshot algorithm.

Our unbounded concurrency atomic write-and-snapshot implementation from registers and two-process consensus objects is given in Algorithm 5. Note that we do not use the

consensus objects directly; rather, they are used to construct n-bounded TAS for infinitely many values of n using the algorithm of [3]. To use such an n-bounded TAS a process needs a name in the range $\{1, \ldots, n\}$. We thus have processes use a finite concurrency renaming algorithm [8] before accessing a TAS object, as explained in detail below.

Algorithm 5 is again based on the idea of processes descending through levels. The intuition behind it is as follows. Processes start the algorithm by obtaining an immediate snapshot, and start to descend from the level (size) of the obtained immediate snapshot. Suppose processes $P_1, \ldots, P_5$ obtain an immediate snapshot $\{P_1, \ldots, P_5\}$, while processes $P_6, \ldots, P_{10}$ each obtain an immediate snapshot of size 10, $\{P_1, \ldots, P_{10}\}$. We would like $P_6, \ldots, P_{10}$ to each return a unique snapshot between the sizes 10 and 6, while the remaining processes should return unique snapshots between the sizes 5 and 1.

We achieve this using the idea from [7] of processes "shepherding tokens". Namely, each process has a *token* that can be placed at each level, signifying the fact that the process is active at (or below) that level. When a process descends from Level L to Level $L-1$, it not only places its own token at Level $L-1$, but also attempts to move the token of every process in its immediate snapshot that has not yet reported its level, to Level $L-1$.

Thus, when a process is slow in posting its obtained immediate snapshot (i.e., in posting its initial level), other participating processes "drag" that slow process' token with them as they descend through the levels. Consequentially, a process P will never descend into the level of an immediate snapshot smaller than its own, as processes that obtain such an immediate snapshot are either seen active at some lower level, or their token is shepherded along with P until P stops its descent and returns them in its snapshot.

In [7] this token placing idea was implemented via a complex read/write protocol; in our context we are able to use TAS objects, thus greatly simplifying the implementation of this step.

The details of the implementation follow. To obtain a write-and-snapshot, a process P starts by obtaining an immediate snapshot IS (Line 1). P then starts to descend through Levels $|IS|, |IS|-1, \ldots, 1$ in an attempt to return a snapshot. (As before, we define the *level* of the snapshot IS to be its size.)

When *entering* in Level L, P posts two things to a SWMR register S_P: (1) that P has entered Level L, and (2) which other processes P believes are in Level L or below (Line 4). We call this set of processes the *tokens* that P carries. P then takes a snapshot of the announcement of every process in IS and calculates the number of tokens at Level L or below (Lines 5-6). If P sees that there are at least L such tokens, it attempts to return these processes as its snapshot as follows. First P tries to win a TAS on T_P^L, an object associated with P's token at Level L (Line 8). Losing this TAS signals to P that some other process has dragged it down to a lower Level and so it shouldn't try to return a snapshot from Level L. If P wins T_P^L, it attempts to win the TAS objects associated with Level L and if it wins, returns the tokens that it saw at Level L (Line 9).

Losing the level's TAS also means that P should descend into Level $L-1$. Before entering Level $L-1$, P tries to drag down with it all the processes that have yet to place their token at some level. To do so, P tries to win T_Q^L for

Algorithm 5 Unbounded concurrency atomic write-and-snapshot (Code of process P)

Shared variables:
- $ImmS$: immediate snapshot object
- T_P^L : for each Level L and process P, a $2L-1$-process TAS object associated with P at Level L
- T^L : for each level L, a $2L-1$-process TAS object associated with Level L
- S_P : for each process P, a register (initially NULL)

procedure WriteAndSnap()

```
    // Obtain an IS and remember it for the entire run:
1:  IS := ImmS.ImmSnap()
2:  S := IS
3:  for L := |S| downto 1
4:      S_P := (L, S)                                   // Post S
5:      snap := snapshot{S_Q | Q ∈ IS}
6:      S^L := ⋃_{(L',S')∈snap with L'≤L} S'
7:      if |S^L| ≥ L then
8:          if TAS(T_P^L) successfully then
9:              if TAS(T^L) successfully return S^L
10:         end if
11:     end if
12:     S := ∅                                          // Rebuild token list
13:     for Q ∈ IS with S_Q empty
14:         if TAS(T_Q^L) successfully
15:         then S := S ∪ {Q}
            // Else, some other process drags Q down
16:         elseif S_Q is empty then S := S ∪ {Q}
17:         end if
18:     end for
19:     S := S ∪ {P}
20:     end for
21: end for
end WriteAndSnap
```

every $Q \in IS$ that has not posted anything to its register S_Q (Lines 13-15). If P loses T_Q^L it checks if this is due to Q posting an announcement and grabbing its token T_Q^L, or whether this is because some other process dragged Q into Level $L-1$. In the latter case, P can still safely add Q to its token list (Line 16). Once P finishes dragging processes with it, P enters the next level and repeats this process.

While the TAS objects associated with Level L are accessed by at most L processes, a process would need to obtain a name for itself in the range $1, \ldots, L$ to use an L-process TAS implementation. Since this is impossible to do without using TAS or some other strong primitive [12], we use $(2L-1)$-process TAS objects and have P perform a read/write renaming algorithm before trying to win a TAS. The renaming and its details are omitted from Algorithm 5.

We now turn to prove the following theorem:

THEOREM 9. *Algorithm 5 is a wait-free unbounded concurrency atomic write-and-snapshot implementation from registers and two-process consensus objects (used to implement* TAS*).*

Based on the correctness of the unbounded concurrency immediate snapshot, the following lemma is immediate:

LEMMA 10. *Algorithm 5 is wait-free.*

The correctness of the implementation follows from the following lemma, by arguments similar to those in [4].

LEMMA 11. *There are at most L process tokens at Level L or below.*

PROOF. We prove that the lemma holds in every finite execution. This implies its correctness for the unbounded concurrency model, since any counter-example to the claim occurs at a finite point in time, where our proof applies.

Let $S_1 \subset \cdots \subset S_r$ be the distinct immediate snapshots obtained by processes in Line 1. We prove by induction. The claim is clearly true for Level $|S_r|$. Inductively, the only way for the claim to be violated is if there are L tokens at Level L or below and no process stops at Level L.

There are two possible cases for what happens in Level L. If every process observes at least L tokens at Line 7, then every process P attempts to `TAS` T_P^L. Since only processes that enter Level L can cause this `TAS` to fail, there must be a process that wins its `TAS` and tries to win T^L. Thus, some process wins T^L and we have a contradiction.

The remaining case is that there are processes that observe $< L$ tokens at Line 7. We denote the tokens posted by a process Q at Level L by S_Q^L. Let Q be the process whose posting of (L, S_Q^L) causes the number of tokens at level L or below to be $\geq L$. Note that for every process P that enters Level L, S_P^L contains the token of every process that starts at a level below L (because if P's token set is not the immediate snapshot obtained at Line 1, then when P descends from Level $L+1$ it either sees the announcement of such a token, or places the token into its token set). Therefore Q must be one of the processes that enter Level L and cannot be a process that starts at a level below L.

Every process that sees (L, S_Q^L) in its snapshot (Lines 5-6) will not attempt to `TAS` T_Q^L. Additionally, any process R whose snapshot in Line 5 precedes the posting of (L, S_Q^L) by Q necessarily observes $< L$ tokens at Level L. Thus $S_Q^L \not\subseteq S_R^L$, implying that Q was active concurrently with R at Level $L+1$. (Either Q won a `TAS` that R lost at Level $L+1$, or Q observed some set to be empty and R then found it to be non-empty.) Hence R cannot observe that S_Q is empty when executing Line 16. Thus R, too, does not attempt `TAS` T_Q^L. It follows that Q wins the `TAS` on T_Q^L and proceeds to `TAS` T^L, so there exists a process that stops at Level L, and we are done. □

5. CONCLUSIONS

We introduced a new dimension into the investigation of Common2: unbounded concurrency.

We presented a wait-free immediate snapshot implementation in the unbounded concurrency model, thereby extending [8], which showed that atomic snapshots and renaming are wait-free implementable in this model.

Moreover, we showed that with one exception, all the basic objects that have n-bounded implementations from 2-consensus objects also have unbounded concurrency implementations from 2-consensus.

We conjecture that the exception, the `swap` object, has no unbounded concurrency implementation and we have some informal indications of it (also applicable to queues), beyond "we tried and failed."

6. REFERENCES

[1] Y. Afek, E. Gafni, J. Tromp, and P. M. B. Vitányi. Wait-free test-and-set. In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG 1992)*, pages 85–94, London, UK, 1992. Springer-Verlag.

[2] Y. Afek and E. Weisberger. The instancy of snapshots and commuting objects. *Journal of Algorithms*, 30(1):68–105, 1999.

[3] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing (PODC 1993)*, pages 159–170, New York, NY, USA, 1993. ACM Press.

[4] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing (PODC 1993)*, pages 41–51, New York, NY, USA, 1993. ACM Press.

[5] M. David. A single-enqueuer wait-free queue implementation. In *Proceedings of the 18th International Conference on Distributed Computing (DISC 2004)*, pages 132–143. Springer, 2004.

[6] M. David, A. Brodsky, and F. E. Fich. Restricted stack implementations. In *Proceedings of the 19th International Conference on Distributed Computing (DISC 2005)*, pages 137–151. Springer, 2005.

[7] E. Gafni. A simple algorithmic characterization of uniform solvability. In *Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS 2002)*, pages 228–237, Washington, DC, USA, 2002. IEEE Computer Society.

[8] E. Gafni, M. Merritt, and G. Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing (PODC 2001)*, pages 161–169, New York, NY, USA, 2001. ACM Press.

[9] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2004)*, pages 206–215, New York, NY, USA, 2004. ACM Press.

[10] M. Herlihy. Impossibility results for asynchronous pram. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1991)*, pages 327–336, New York, NY, USA, 1991. ACM Press.

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

[12] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.

[13] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[14] Z. Li. Non-blocking implementations of queues in asynchronous distributed shared-memory systems. Master's thesis, Department of Computer Science,

University of Toronto, 2001.
[15] M. Merritt and G. Taubenfeld. Computing with infinitely many processes. In *Proceedings of the 14th International Conference on Distributed Computing (DISC 2000)*, pages 164–178, London, UK, 2000. Springer-Verlag.

APPENDIX

A. STACK LINEARIZABILITY PROOF

This section is devoted to the proof of the following theorem.

THEOREM 12. *Algorithm 1 is a linearizable* `stack` *implementation from read/write registers,* `swap` *and* `F&A`.

We construct a linearization S of H, an arbitrary history of our stack implementation. Each linearized high-level operation *op* in S has an associated linearization point, denoted LinPt(*op*), which is the event in H at which *op* appears to take effect. As distinct operations may be linearized next to each other at the same event, S is provided explicitly and dictates the order of such operations.

Before proceeding with the proof, we formally state our model and notation.

A.1 Model and notations

An execution in the system is modeled by a *history*, which is a (possibly infinite) sequence of (base and high-level) invocation and response and events. Each event includes the identity of the object and the process performing the operation. An *operation op* in a history is a pair consisting of an invocation, $inv(op)$, and the first response with the same object and process id following the invocation, $res(op)$. Each base object invocation is immediately followed by its response (these operations are *atomic*) and so we write them in pseudo-code form.

A history H induces a partial order $\prec_H$ on operations: $op_0 \prec_H op_1$ if $res(op_0)$ precedes $inv(op_1)$ in H. If op_0 and op_1 are incomparable by $\prec_H$ we say that they are *concurrent*. A *sequential* history begins with an invocation and alternates matching invocations and responses (notice that if H is sequential then $\prec_H$ is a total order). An operation is *active* in H if only its invocation appears in H but its response does not.

A *process subhistory* of H is the subsequence of H containing all the events performed by P and is denoted $H|P$. Two histories H and H' are *equivalent* if $H|P = H'|P$ for every process P. The subsequence of H consisting of all matching invocations and responses is denoted complete(H). H is *linearizable* if it can be extended (by adding response events) into a history H' such that: (1) complete(H') is equivalent to some legal sequential history S; (2) $\prec_H \subseteq \prec_S$. Such an S is called the *linearization* of H. An implementation is *correct* if for every history H of the implementation, the *object subhistory* of O (defined similarly to a process history) is linearizable.

A.2 Eliminating easily linearizable operations

In this subsection we explain how to eliminate from H the concurrent pairs of Push(x)/Pop($\uparrow x$) operations. Simply deleting the events of a concurrent (Push(x), Pop($\uparrow x$)) pair from H would leave us with an illegal history, in which `range` never returns $index(x)$ but possibly returns larger values. To overcome this technicality, we modify the stack specification to include a Dummy-Push operation that does not change the stack's state and is implemented by a single increment of `range`. Now we delete just the `write` event of Push(x) and all the events of Pop($\uparrow x$), and think of Push(x) as a Dummy-Push(x) operation. Since Dummy-Push operations do not affect the stack, they can be linearized at any point in time and so we ignore the issue of linearizing them. We thus prove that the modified stack (with Dummy-Push operations) is correct, which also proves that the original implementation is correct.

Formally, define Conc[H] to be the set of concurrent pairs (Push(x), Pop($\uparrow x$)) in H. For $C \subseteq$ Conc[H], define H *with* C *eliminated*, denoted $H \setminus C$, as the history obtained from H by eliminating each (Push(x), Pop($\uparrow x$)) $\in C$ as described above. The following lemma states that $H \setminus C$ is a history just like H, except that in $H \setminus C$ the primitive operations performed by Push operations from C are carried out by Dummy-Push operations that terminate without writing to `items`.

LEMMA 13. *$H \setminus C$ is a history in the (modified) stack implementation, and $\prec_{H\setminus C} \subseteq \prec_H$.*

The following lemma shows that given a linearization of $H \setminus C$, we can linearize all the removed operations at the points in H where they were both active.

LEMMA 14. *Suppose S (with associated LinPt(·)) is a linearization of $H \setminus C$ that does not include any Push operation from C. Then H is linearizable.*

PROOF. We use induction on $|C|$. If $C = \emptyset$ then S is a linearization of H and the claim is immediate. For the inductive case, obtain a set C' by removing some pair (Push(x), Pop($\uparrow x$)) from C, and notice that the linearization of $H \setminus C$ can be turned into a linearization of $H' = H \setminus C'$ by discarding the linearization of the Dummy-Push(x) operation and linearizing Push(x) immediately followed by Pop($\uparrow x$) at an event $e \in H'$ where both of them are active. □

A.3 Linearizing the reduced history

Henceforth, following the previous subsection, we consider only H'— a stack history in which for any value v, Push(v) and Pop($\uparrow v$) are not concurrent.

In this subsection we construct S, the linearization of the completed high level operations in history H' (see Algorithm 6 for the construction procedure **L**). In the construction each high level operation *op* whose response event is in H' is linearized at some event in H' which is between *op*'s invocation and response events. Procedure **L** that constructs S, processes the events in H' one after the other according to their order in H'. It uses an auxiliary array I of infinite size that represents the stack state at each point during S and mimics the operations of the implementation in array `items`. The stack state represented by I consists of the non-NULL values stored in I according to their order in the array. We denote by $top(I)$ the value in the highest index non-NULL cell of I.

When processing an event e, if e is the write event of a Push(x) operation (Line 2 in Algorithm 1), **L** places x into $I[index(x)]$ and then decides where to linearize Push(x). If $x = top(I)$, **L** appends Push(x) to S, and e is assigned

as LinPt(Push(x)). Otherwise, **L** linearizes Push(x) in S immediately before LinPt(Push(y)), where y is the lowest non-NULL item above x in I.

For any event e of H' processed by **L**, an attempt is made to linearize Pop($\cdot$) operations at e as follows. If the Pop($\uparrow$ *top*(I)) operation is active at e, **L** defines LinPt(Pop($\uparrow$ *top*(I))) to be e and sets the current *top*(I) cell in I to NULL. This is repeated until Pop($\uparrow$ *top*(I)) is not active, and **L** then proceeds to the next event in H'.

In the rest of the proof we show that S is a legal history according to the stack specification (Lemma 15). Lemmas 16 and 19 show that each high-level operation is linearized at some point of time (an event in H') between its invocation and response. In the following we denote by I_σ and S_σ the values of I and S after processing prefix σ of H'. When the context is clear, we drop the subscript.

LEMMA 15. *S is a legal stack history.*

PROOF. We use induction on σ to show that for every prefix σ of H', S_σ is a legal stack history and I_σ is the content of the linearized stack at S_σ. The claim is clearly true for $\sigma = \varepsilon$. For the inductive step, we have $\sigma = \sigma_1\ e$ and $I_{\sigma_1} = [x_m, \ldots, x_0 = \perp]$.

If $e =$<`items[i] := x`>, **L** executes lines 2-10 in Algorithm 6. If **L** appends Push(x) to S, then $x = top(I)$. Thus, S correctly reflects the value of $I = [x, x_m, \ldots, \perp]$. Otherwise, **L** finds that the first non-NULL value above cell i is some x_k. Because x_k is currently in I, it follows from the induction hypothesis that placing Push(x) immediately before Push(x_k) in S produces a valid stack history that is consistent with $I = [x_m, \ldots, x_k, x, x_{k-1}, \ldots, \perp]$.

L subsequently executes lines 11-15, where Pop operations may be appended to S. Whenever Pop($\uparrow v$) is appended to S, $v = top(I)$ and so by the induction hypothesis, at that moment S is a legal stack history that is consistent with I. □

For the proof of the following lemma it is convenient to assume that in H', the invocation event for every high-level operation *op* is immediately before the first primitive operation in the implementation of *op*, and similarly that the response event of *op* is immediately after the last primitive operation of *op*. Therefore, throughout the proof, we treat the corresponding primitive events as the high-level invocations and responses events. That is:

High level event	**Primitive event**	
Push invocation	`F&A(range, 1)`	(Line 1)
Push(x) response	`items[i] := x`	(Line 2)
Pop invocation	read of `range`: `F&A(range, 0)`	(Line 3)
Pop($\uparrow v$) response	`swap(items[i], NULL)` that returns v	(Line 4)
Pop($\uparrow\perp$) response	`swap` on cell 1 that returns `NULL`	(Line 4)

LEMMA 16. *Let $\sigma = \sigma' e$ be a prefix of H' such that e is a response of high-level operation op, then LinPt(op) is in S_σ.*

PROOF OF LEMMA 16. The claim clearly holds for Push operations as they take effect upon their write to `items`. The linearization point of a Pop operation is an event where that operation is active. So if the claim is false, there must be a shortest prefix $\bar\sigma$ of H' ending with a response event $\bar e$ of Pop($\uparrow v$) (possibly $v = \perp$) such that LinPt(Pop($\uparrow v$)) is not defined. Before completing the proof, we need the following two claims:

CLAIM 17. *For any prefix δ of $\bar\sigma$, if $x \in I_\delta$ then $x \in$* `items` *at δ.*

PROOF OF CLAIM 17. Since **L** adds x to I only upon processing the write of x to `items`, x must have been written to `items` during δ. If x has been removed from `items` before δ (i.e., the response of Pop($\uparrow x$) occurs in δ and x is still in I_δ), then Pop($\uparrow x$) is a Pop operation that terminated without being linearized. This means that LinPt(Pop($\uparrow x$)) is not defined when Pop($\uparrow x$) terminates in δ, contradicting our assumption that $\bar\sigma$ is the shortest such run. □

CLAIM 18. *Suppose $\bar\sigma = \delta\ \gamma\ e\ \beta$, where (1) $x \neq v$, $x \in I_\delta$, and (2) e is a* `swap` *on index(x) by Pop($\uparrow v$). Then LinPt(Pop($\uparrow x$)) is in γ.*

PROOF OF CLAIM 18. By Claim 17, x is in `items` at δ. Since $x \neq v$, Pop($\uparrow v$)'s `swap` on *index*(x) returns `NULL`, implying that Pop($\uparrow x$) `swap`s x in γ. Since the `swap` that returns x is the response event of Pop($\uparrow x$), then by the assumption on $\bar\sigma$, LinPt(Pop($\uparrow x$)) is in γ. □

Continuing the proof that **L** must have linearized Pop($\uparrow v$) in σ, let $\bar\sigma = \delta_1\ \gamma$, such that the last event of δ_1 is Pop($\uparrow v$)'s read of the `range` register (i.e., Pop($\uparrow v$)'s invocation). Because H' does not contain easily linearizable pairs, $v \in I_{\delta_1}$. However, since Pop($\uparrow v$) is not linearized at δ_1, $top(I_{\delta_1}) \neq v$. Let x_1 denote $top(I_{\delta_1})$. By Claim 17, x_1 is in `items` at δ_1, and Pop($\uparrow v$) must have read a value $r > index(x_1)$ from `range`. Therefore, Pop($\uparrow v$) performs a `swap` on *index*(x_1) in γ.

To complete the proof of the Lemma we show that if the conditions required to linearize Pop($\uparrow v$) do not materialize before the response event of Pop($\uparrow v$) then Pop($\uparrow v$) must perform a `swap` on an infinite sequence of elements between x_1 and v. In other words, since the conditions (Line 11 in Algorithm 6) do not hold there must be an element such as x_1 covering v, then we show that x_1 must be swapped out thus either exposing v and enabling the linearization of Pop($\uparrow v$), or there must be another element x_2 under x_1 and above v, and so on. This leads to a contradiction, since Pop($\uparrow v$) performs only a finite number of steps.

Formally, we prove that there is an infinite sequence of length-increasing prefixes of $\bar\sigma$, $\{\delta_i\}_{i=1}^{\infty}$. We show that each δ_i in the sequence has the following two properties: (1) let $x_i = top(I_{\delta_i})$, then $x_i \neq v$, and (2) Pop($\uparrow v$) performs a `swap` on *index*(x_i) after δ_i. Above we showed that δ_1 has these properties. Next, we provide a general proof that given a prefix δ_i of $\bar\sigma$ with properties (1) and (2), there exists a longer prefix δ_{i+1} of $\bar\sigma$ with the same properties.

By property (2) of δ_i, Pop($\uparrow v$) performs a `swap` on *index*(x_i) after δ_i. Thus, by Claim 18, $\bar\sigma = \delta_i\ \alpha\ \gamma'$ where Pop($\uparrow v$) performs the `swap` on *index*(x_i) in γ', and Pop($\uparrow x_i$) is linearized at the last event of $\delta_{i+1} = \delta_i\ \alpha$. Since Pop($\uparrow v$) is not linearized at δ_{i+1}, $top(I_{\delta_{i+1}}) \neq v$. This proves property (1) for δ_{i+1}.

To prove property (2) for δ_{i+1}, let x_{i+1} denote $top(I_{\delta_{i+1}})$. By Claim 17, x_{i+1} is in `items` at δ_{i+1}. In addition, $index(x_{i+1}) <$

Algorithm 6 Procedure for linearizing H'

```
   Variables:
       I : array of values, initially I[0] = ⊥ and all other cells are NULL
       S : history, initially ε

   procedure L(H' = e_1 ...)
1:     for j := 1 to ...
2:         if e_j = <items[i] := x>   // Line 2 Algorithm 1
3:             I[i] := x
4:             let y be the lowest non-NULL item stored above x in I
5:             if no such y exists then S := S Push(x)   // LinPt(Push(x)) = e_j
6:             else
7:                 let S = S_1  Push(y)  S_2
8:                 S := S_1  Push(x) Push(y)  S_2   // LinPt(Push(x)) = LinPt(Push(y))
9:             end if
10:        end if
11:        while Pop(↑ top(I)) is active in e_1 ... e_j
12:            let v = top(I)
13:            S := S  Pop(↑ v)      // LinPt(Pop(↑ v)) = e_j
14:            if v ≠⊥ then I[index(v)] := NULL
15:        end while
16:    end for
   end L
```

$index(x_i)$ or Pop($\uparrow$ x_i) would not have been linearized at δ_{i+1}. Since Pop($\uparrow$ v) performs its `swap` on $index(x_i)$ in γ', it can only perform the `swap` on $index(x_{i+1})$ after δ_{i+1}. □

LEMMA 19. *Let $H' = \alpha\ \sigma\ \beta$ such that the first event of σ is the invocation of a high-level operation op, and the last event of σ is the response of op. Then LinPt(op) is defined during σ.*

PROOF. By Lemma 16, LinPt(op) is defined. Thus, the claim is immediate for Pop operations and for Push operations that are linearized at their write event.

The remaining case is that of a Push operation which is linearized in Line 7 of **L**. Let Push(x) be such an operation. Then there is a sequence Push(x_1), ..., Push(x_k) of Push operations whose linearization point is LinPt(Push(x)) and that appear after Push(x) in S. Thus, LinPt(Push(x)) is the write event of Push(x_k). We proceed to show that Push(x) is active when Push(x_k) performs its write event.

Since $index(x) < index(x_k)$, Push(x) must have performed its `F&A` before Push(x_k), and so Push(x) is active before the write event of Push(x_k). In addition, the fact that **L** processed Push(x_k)'s write event before processing Push(x)'s write event means that Push(x) is active at LinPt(Push(x)), and we have the claim. □