Nontrivial and Universal Helping for Wait-Free Queues and Stacks

Hagit Attiya*[1], Armando Castañeda†[2], and Danny Hendler[3]

1 Technion – Israel Institute of Technology, Haifa, Israel
2 Universidad Nacional Autónoma de México (UNAM), Mexico City, Mexico
3 Ben-Gurion University, Beer-Sheva, Israel

Abstract

This paper studies two approaches to formalize helping in wait-free implementations of shared objects. The first approach is based on *operation valency*, and it allows us to make the important distinction between *trivial* and *nontrivial* helping. We show that a wait-free implementation of a queue from common2 objects (e.g., Test&Set) requires nontrivial helping. In contrast, there is a wait-free implementation of a stack from Common2 objects with only trivial helping. This separation might shed light on the difficulty of implementing a queue from Common2 objects.

The other approach formalizes the helping mechanism employed by Herlihy's universal wait-free construction and is based on having an operation by one process restrict the possible linearizations of operations by other processes. We show that objects possessing such *universal helping* can be used to solve consensus.

1998 ACM Subject Classification C.1.4 Parallel Architectures, D.4.1 Process Management

Keywords and phrases helping, wait-free, nonblocking, queues, stacks, common2

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.31

1 Introduction

A key component in the design of concurrent applications are *shared objects* providing higher-level semantics for communication among processes. For example, a *shared queue* to which processes can concurrently enqueue and dequeue, allows them to share tasks, and similarly a *shared stack*. Shared objects are implemented from more basic *primitives* supported by the multiprocessing architecture, e.g., reads, writes, Test&Set, or Compare&Swap. An implementation is *wait-free* if an operation on the shared object is guaranteed to terminate after a finite number of steps; the implementation is *nonblocking* if it only ensures that *some* operation (perhaps by another process) completes in this situation. Clearly, a wait-free implementation is nonblocking but not necessarily vice versa.

Many implementations of shared objects, especially the wait-free ones, include one process *helping* another process to make progress. The helping mechanism is often some code that is added to a nonblocking implementation. Typically, the code uses only reads and writes, in addition to the primitives used in the nonblocking implementation (e.g., Test&Set). The aim of this extra code is that processes that complete an operation "help" the blocked processes to terminate, so that the resulting implementation is wait-free. An interesting

* Supported by the Israel Science Foundation (grant 1749/14).
† Supported partially by PAPIIT-UNAM IA101015. This research was partially done while the second author was at the Department of Computer Science of the Technion, supported by an Aly Kaufman post-doctoral fellowship.

© Hagit Attiya, Armando Castañeda, and Danny Hendler;
licensed under Creative Commons License CC-BY

example is a shared queue, for which there is a simple nonblocking implementation using only reads, writes and Test&Set [14]. Such a helping mechanism would provide a wait-free queue implementation using those primitives, showing that the queue belongs to Common2, the family of shared objects that are wait-free implementable from primitives with consensus number 2 for any number of processes [1, 2]. The Common2 family contains Test&Set, Swap, stacks and other objects.

The question whether queues belong to Common2 has been open for many years and has received a considerable amount of attention [2, 4, 5, 6, 8, 14]. It essentially asks if there is an n-process linearizable wait-free implementation of a queue from Test&Set, for every $n \geq 3$.

This paper investigates ways to formalize helping, with the purpose of being able to separate objects for which there are wait-free implementations from those with only nonblocking implementations. We are especially interested in implementations using primitives with finite *consensus number* [12], like Test&Set, which allows us to solve consensus exactly for two processes. Primitives with an infinite consensus number, like Compare&Swap, are universal and provide generic wait-free implementation for any shared object [12]; clearly, with such primitives nonblocking and wait-freedom cannot be separated.

We first introduce a notion of helping that is based on one process determining the return value of an operation by another process; it relies on the notion of *operation valency* [11], i.e., the possible values an operation might return. Roughly speaking, an implementation has helping if in some situation, a process makes an undecided operation of another process become decided on some value. In the context of specific objects, like queues and stacks, which have a distinguished "empty" value (denoted $\perp$), we say that helping is *nontrivial* if one process makes another become decided on a non-$\perp$ value. Helping is nontrivial since the helping process needs to "grab" the value it gives to the helped process. Therefore, the helping process has to communicate with the other processes to ensure that this value is not taken by someone else.

Our first main result is a separation between stacks and queues implemented from Test&Set. It shows that any wait-free queue must have nontrivial helping while this is not true for stacks, as we show that the wait-free stack of Afek et al. [1] (which established that stacks belong to Common2) does not have nontrivial helping.

The paper also studies an alternative way to formalize helping, which is based on restricting the possible linearizations of an operation by the progress of another process. This kind of helping, which we call *universal*, formalizes the helping mechanism employed in Herlihy's universal construction. Intuitively, an implementation has universal helping if for every execution α, for every long enough extension of it, all pending operations in α (which might be still pending in the extension) are linearized. We show that universal helping for queues and stacks is strong enough to solve consensus, namely, any wait-free n-process implementation of a queue or stack with universal helping can solve n-process consensus.

These results provide insights on why finding a wait-free implementation for queues from Test&Set has been a longstanding open question: any such implementation must have some helping mechanism; however, this mechanism cannot be too strong, otherwise the resulting implementation would be able to solve consensus for n processes, $n \geq 3$, which is impossible to do with Test&Set since it has consensus number 2.

Finally, the paper compares the two formalizations proposed here to the formalization of helping recently introduced in [3]. It also shows that universal helping has implications on *strong linearizability* [9] for queues and stacks: there is no n-process wait-free strong linearizable implementation of queues or stacks from primitives with consensus number smaller than n.

2 Model of Computation

We consider a system with n *asynchronous* processes, $p_1, \ldots, p_n$. Processes communicate with each other by applying *primitives* to shared *base objects*; the primitives can be read and write, or more powerful primitives like Test&Set or Compare&Swap. Any process may crash at any time in an execution, namely, it stops taking steps from that point on. A process that does not crash is *correct*.

A *(high-level) concurrent object*, or *data type*, is defined by a state machine consisting of a set of states, a set of operations, and a set of transitions between states. Such a specification is known as *sequential*. In the rest of the paper we will concentrate on stacks and queues. A *shared stack* provides two operations push($\cdot$) and pop(). A push(x) operation puts x at the top of the stack, and a pop() removes and returns the value at the top, if there is one, otherwise it returns $\perp$. A *shared queue* provides operations enq($\cdot$) and deq(). An enq(x) operation puts x at the tail of the queue, and a deq() removes and returns the value at the head of the queue, if there is one, otherwise it returns $\perp$.

An *implementation* of an object O is a distributed algorithm $\mathcal{A}$ consisting of local state machines $A_1, \ldots, A_n$. Local machine A_i specifies which primitives p_i executes in order to return a response when it invokes an operation of O. An implementation is *wait-free* if every process completes each of its invocations in a finite number of its steps. Formally, if a process executes infinitely many steps in an execution, it completes all its invocations. An implementation is *nonblocking* if whenever processes take steps, at least one of the operations terminates. Namely, in every infinite execution, infinitely many invocations are completed. Thus, a wait-free implementation is nonblocking but not necessarily vice versa.

A *configuration* C of the system is a collection containing the states of all base objects and processes. A configuration is *initial* if base objects and processes are in initial states. Given a configuration C, for any process p, $p(C)$ denotes the configuration after p takes its next step. A process p is *idle* in a configuration C if p is in a state in which all its operations are completed.

An *execution* of the system is modelled by a *history*, which is a possibly infinite sequence of invocations and responses of high-level operations and primitives. For a set of processes S, an S*-execution* is an execution in which only processes in S take steps. If $S = \{p\}$, we say that the execution is p*-solo*. An operation op in a history is *complete* if both its invocation $inv(\mathsf{op})$ and response $res(\mathsf{op})$ appear in the history. An operation is *pending* if only its invocation appears in the history.

A history H induces a natural partial order $<_H$ on the operations of H: $\mathsf{op} <_H \mathsf{op}'$ if and only if $res(\mathsf{op})$ precedes $inv(\mathsf{op}')$. Two operations are *concurrent* if they are incomparable. A *sequential* history alternates matching invocations and responses and starts with an invocation event. Hence, if H is sequential, $<_H$ induces a total order.

Linearizability [13] is the standard notion used to identify a correct implementation. Roughly speaking, an implementation is linearizable if each operation appears to take effect atomically at some time between the invocation and response of an operation.

Let $\mathcal{A}$ be an implementation of an object O. A history H of $\mathcal{A}$ is *linearizable* if H can be extended by adding response events for some pending invocations such that the sequence H' containing only the invocation and responses of O agrees with the specification of O, namely, there is an initial state of O and a sequence of invocations and responses that produces H'. We say that $\mathcal{A}$ is *linearizable* if each of its histories is linearizable.

In the *consensus* problem, each process proposes a value and is required to decide on a value such that the following properties are satisfied in every execution:

Termination. Every correct process decides.
Agreement. Processes decide on the same value.
Validity. Processes decide proposed values.

Consensus is *universal* [12] in the sense that from reads and writes and objects solving consensus among n processes, it is possible to obtain a wait-free implementation for n processes of any concurrent object with a sequential specification. The *consensus number* of a primitive [12] is the maximum number n such that it is possible to solve consensus on n processes from reads, writes and the primitive. For example, the consensus number of Test&Set is 2. Hence, Test&Set allows us to implement any concurrent object in a system with 2 processes.

3 Separating Stacks and Queues with Nontrivial (Valency-Based) Helping

In this section, we present a notion of helping that differentiates between queues and stacks: any queue implementation must exhibit this kind of helping, but there is a stack implementation that does not (essentially, that of [1]). This sheds some light on the difficulty of finding a wait-free implementation of a queue from Common2.

Let $\mathcal{A}$ be a wait-free linearizable implementation of a data type T, such as a stack or queue. The input for an invocation of an operation of T is from some domain $\mathcal{V}$ and the output of a response is from the domain $\mathcal{V} \cup \{\perp\}$, where $\perp \notin \mathcal{V}$ denotes the *empty* or *initial* state of T.

Let C be a reachable configuration of $\mathcal{A}$ and let $\mathsf{opType}(\cdot)$ be an operation by a process p. We say that $\mathsf{opType}(\cdot)$ is *v-univalent in C* (or just *univalent* when v is irrelevant) if in every configuration C' that is reachable from C in which $\mathsf{opType}(\cdot)$ is complete, its output value is v; otherwise, $\mathsf{opType}(\cdot)$ is *multivalent in C*. We say that $\mathsf{opType}(\cdot)$ is *critical on v in C* (or just *critical in C*) if it is multivalent in C but v-univalent in $p(C)$.

▶ **Definition 1** (Nontrivial and trivial (valency-based) helping)**.** Process q *helps* process $p \neq q$ in configuration C if there is a multivalent $\mathsf{opType}(\cdot) \in C$ by p that is v-univalent in $q(C)$. We say that q *nontrivially helps* p if $v \neq \perp$; otherwise, it *trivially* helps p.
An implementation of a type T has *nontrivial (trivial) helping* if it has a reachable configuration C such that some process q nontrivially (trivially) helps process p in C.

Directly from the previous definition we get the following claim.

▶ **Claim 2.** *If C is a reachable configuration of an algorithm without nontrivial helping, and an operation* op *by p is multivalent in C, then* op *is not v-valent in $q(C)$, for any value $v \neq \perp$ and process $q \neq p$.*

The proof of the next theorem captures the challenging "tail chasing" phenomenon one faces when trying to implement a queue from objects in Common2. Observe that in the case of a queue implementation, only dequeues can be nontrivially helped since enqueues always return true, and are therefore trivially univalent.

▶ **Theorem 3.** *Any two-process wait-free linearizable queue implementation from* read/write *and* Test&Set *operations has nontrivial helping.*

Proof. Assume, by way of contradiction, there is such an implementation $\mathcal{A}$ without nontrivial helping. Let p and q be two distinct processes, and let C_{init} be the initial configuration of $\mathcal{A}$.

For any $k \geq 1$, we construct an execution α_k of p and q, starting with C_{init} and ending in configuration C_k. In α_k, p executes a single $\mathsf{deq_p}()$ operation, and the following properties hold:

1. q is idle in C_k,
2. p has at least k steps in α_k,
3. in every linearization of α_k, all enqueues appear in the same order and enqueue distinct values,
4. there is no linearization of α_k in which $\mathsf{deq_p}()$ outputs $\perp$, and
5. $\mathsf{deq_p}()$ is multivalent in C_k (in particular, it is pending).

We proceed by induction. For the base case, $k = 1$, let α_1 be the execution that starts at C_{init} and in which p completes alone $\mathsf{enq}(1)$ and then starts $\mathsf{deq_p}()$ until it is critical on 1. This execution exists because $\mathcal{A}$ is wait-free. Clearly, there is no linearization of α_1 in which $\mathsf{deq_p}()$ outputs $\perp$. The other properties also hold.

Suppose that we have constructed α_k, $k \geq 1$; we show how to obtain α_{k+1}. Let β^1 be the q-solo extension of α_k in which q completes $\mathsf{enq}(z)$, where z is a value that is not enqueued in α_k, and then starts a $\mathsf{deq}_q()$ operation. Let β^2 be an extension of $\alpha_k\,\beta^1$ in which p and q take steps until both their dequeue operations are critical. The extension β^2 exists because, first, $\mathcal{A}$ is wait-free and, second, by Claim 2, a step of p does not make $\mathsf{deq_q}()$ univalent, and a step of q does not make $\mathsf{deq_p}()$ univalent.

Let C be the configuration at the end of $\alpha_k\,\beta^1\,\beta^2$; note that $\mathsf{deq_p}()$ is critical on some value y_p in C and that $\mathsf{deq_q}()$ is critical on some value y_q in C.

Note that neither y_p nor y_q is $\perp$ since the queue has at least two values in C. This holds since the induction hypothesis is that there is no linearization of α_k in which $\mathsf{deq_p}()$ outputs a non-$\perp$ value, and in the extension $\beta^1\,\beta^2$, q first completes an enqueue and then starts a dequeue.

By a similar argument, there is no linearization of $\alpha_k\,\beta^1\,\beta^2$ in which either $\mathsf{deq_p}()$ or $\mathsf{deq_q}()$ outputs $\perp$.

The following claim is where the specification of a queue comes into play. (This claim does not hold for a stack, for example.)

▶ **Claim 4.** $y_p = y_q$.

Proof. Suppose, by way of contradiction, that $y_p \neq y_q$. By the induction hypothesis, in every linearization of α_k, the order of enqueues is the same. The same holds for $\alpha_k\,\beta^1\,\beta^2$ because q is idle in α_k, by induction hypothesis, and then its enqueue in β^1 happens after all enqueues in α_k. Suppose, without loss of generality, that $\mathsf{enq}(y_q)$ precedes $\mathsf{enq}(y_p)$ in every linearization of $\alpha_k\,\beta^1\,\beta^2$. Consider the p-solo extension in which p completes $\mathsf{deq_p}()$, ending with configuration D.

Since $\mathsf{deq_p}()$ is critical on y_p in C, it outputs y_p in D. We claim that $\mathsf{deq_q}()$ outputs y_q in every extension from D. Otherwise, another dequeue outputs y_p in some extension from D. Since this dequeue starts after $\mathsf{deq_p}()$ completes, it must be linearized after $\mathsf{deq_p}()$. This contradicts the linearizability of $\mathcal{A}$, since in every linearization of $\alpha_k\,\beta^1\,\beta^2$, $\mathsf{enq}(y_q)$ precedes $\mathsf{enq}(y_p)$. Therefore, $\mathsf{deq_q}()$ is y_q-univalent in D. This contradicting Claim 2, since a step of p makes $\mathsf{deq_q}()$ univalent on a non-$\perp$ value. ◀

Note that there is no extension of C in which $\mathsf{deq_p}()$ and $\mathsf{deq_q}()$ output the same value because the enqueues in α_k have distinct values and in β^1, q enqueues z, a value that is not enqueued in α_k.

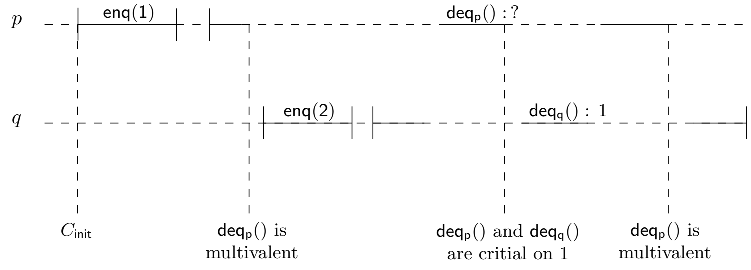

Figure 1 Getting α_2 from α_1.

Assume that p is poised to access R_p in C (i.e. the next step of p is on R_p) and that q is poised to access R_q in C. If $R_p \neq R_q$, then in the p-solo extension of $q(p(C))$ in which p completes $\mathsf{deq_p}()$, its output is $y = y_p = y_q$. But in $p(q(C))$, the local state of p and the state of the shared memory is the same as in $q(p(C))$. Hence, in the p-solo extension of $p(q(C))$, p completes $\mathsf{deq_p}()$ with output y, as well. This contradicts the fact that $\mathsf{deq_q}()$ is critical on y in C. Thus, $R_p = R_q = R$.

A similar argument, by case analysis, shows that p and q must apply Test&Set primitives to R in C, and that the value of R is 0 in C. These facts are used in the proof of the next claim.

▶ **Claim 5.** $\mathsf{deq_p}()$ *is not critical in* $q(C)$.

Proof. Let $y = y_p = y_q$ be the value that $\mathsf{deq_p}()$ and $\mathsf{deq_q}()$ are critical on in C. Suppose, by way of contradiction, that $\mathsf{deq_p}()$ is critical on y' in $q(C)$. We have that $y' \neq y$.

Let γ be an extension of $\alpha_k\,\beta^1\,\beta^2\,q$ in which $\mathsf{deq_p}()$ outputs. Write $\gamma = \lambda^1\,p\,\lambda^2$, where λ^1 is p-free (λ^1 might be empty). Since p and q are about to perform Test&Set primitives on R in C, the state of the shared memory and the local state of p are the same at the end of $\alpha_k\,\beta^1\,\beta^2\,q\,\lambda^1\,p$ and $\alpha_k\,\beta^1\,\beta^2\,q\,p\,\lambda^1$, because in both executions q is the first process accessing R (using Test&Set) and then when p accesses R (using Test&Set also), it gets false, no matter when it accesses R. Then, p is in the same local state in $\alpha_k\,\beta^1\,\beta^2\,q\,\lambda^1\,p\,\lambda^2$ and in $\alpha_k\,\beta^1\,\beta^2\,q\,p\,\lambda^1\,\lambda^2$. We have that $\mathsf{deq_p}()$ is critical in $q(C)$, which implies that the output of it in $\alpha_k\,\beta^1\,\beta^2\,q\,p\,\lambda^1\,\lambda^2$ is y', and thus the output of $\mathsf{deq_p}()$ in $\alpha_k\,\beta^1\,\beta^2\,q\,\lambda^1\,p\,\lambda^2$ is y' too, since, as already said, the local state of p is the same in both executions. This implies that $\mathsf{deq_p}()$ is univalent in $q(C)$, contrary to our assumption that it is critical in $q(C)$. ◀

Let $\alpha_{k+1} = \alpha_k\,\beta^1\,\beta^2\,q\,p\,\beta^3$, where β^3 is the q-solo extension in which q completes its $\mathsf{deq}_q()$ (β^3 exists because $\mathcal{A}$ is wait-free). See Figure 1. We argue that α_{k+1} has the desired properties.

1. q is idle in α_{k+1} because in β^3 it completes $\mathsf{deq_q}()$ and does not start a new operation.
2. p has at least $k+1$ steps of $\mathsf{deq_p}()$ in α_{k+1}, since p has at least k steps of $\mathsf{deq_p}()$ in α_k, by the induction hypothesis, and at least one step in $\beta^1\,\beta^2\,q\,p\,\beta^3$.
3. By the induction hypothesis, in every linearization of α_k, the enqueue operations follow the same order. The enqueue of q in β^1 happens after all enqueues in α_k. Then, in every linearization of α_{k+1}, the enqueues follow the same order. The enqueues in α_{k+1} enqueue distinct values because that is true for α_k, by the induction hypothesis, and the enqueue of q in β^1 enqueues a value that is not in α_k.
4. As argued above, there is no linearization of $\alpha_k\,\beta^1\,\beta^2$ in which either $\mathsf{deq_p}()$ or $\mathsf{deq_q}()$ output $\perp$. In β^3, q just completes $\mathsf{deq_q}()$. Then, there is no linearization of α_{k+1} in which $\mathsf{deq_p}()$ outputs $\perp$.

```
Shared Variables:
    range : Fetch&Add register initialized to 1
    items : array [1,...,] of read/write registers
    T : array [1,...,] of Test&Set registers

Operation Push(x):
(01)  i = Fetch&Add(range, 1)
(02)  items[i] ← x
      return true
end Push

Operation Pop():
(03)  t = Fetch&Add(range, 0)
      for i ← t downto 1 do
(04)      x ← items[i]
(05)      if x ≠ ⊥ then
(06)          if Test&Set(T[i]) then return x
          end if
      end for
(07)  return ⊥
end Pop
```

Figure 2 The stack implementation of Afek et al. [1].

5. Since $\text{deq}_p()$ is not critical in $q(C)$, it is multivalent at the end of $\alpha_k\,\beta^1\,\beta^2\,q\,p$. Since β^3 is q-solo, Claim 2 implies that $\text{deq}_p()$ is multivalent at the end of α_{k+1}.

This yields an execution of $\mathcal{A}$ in which p executes an infinite number of steps but its $\text{deq}_p()$ operation does not complete, contradicting the wait-freedom of $\mathcal{A}$. ◀

As we just saw, any wait-free implementation of a queue from Test&Set must have nontrivial helping. This is not the case for stack implementations, as we show next.

▶ **Theorem 6.** *There is an n-process wait-free linearizable stack implementation from read/write and m-process Test&Set primitives, $2 \leq m \leq n$, without nontrivial helping.*

Proof. First, we show that an n-process wait-free linearizable Test&Set operation can be implemented from 2-process Test&Set without nontrivial helping. [2] present an n-process wait-free linearizable implementation of a Test&Set operation from 2-process one-shot swap primitives. Let us call this algorithm $\mathcal{A}$. It is easy to check that $\mathcal{A}$ does not have nontrivial helping. It is also easy to implement one-shot 2-process swap from 2-process Test&Set without helping (the processes just use Test&Set to decide who swaps first), and hence, from $\mathcal{A}$ we can get an n-process wait-free linearizable implementation of a Test&Set operation from 2-process Test&Set without nontrivial helping. Let us call the resulting algorithm $\mathcal{B}$.

Now, consider Afek et al.'s stack implementation [1] (Figure 2). We argue that the implementation does not have nontrivial helping: just note that there is no configuration C in which process q makes another process p v-univalent, $v \neq \perp$, because the only way a pop operation becomes univalent on a non-$\perp$ value is by winning the Test&Set in line 6; thus, it is impossible that a multivalent pop operation by p in C becomes univalent on a non-$\perp$ value in $q(C)$, with $q \neq p$.

Afek et al. proved that one can get an n-process wait-free linearizable Fetch&Add from 2-process wait-free linearizable Test&Set primitives [2]. Let $\mathcal{C}$ be this implementation.

Now, we replace each Test&Set primitive in Afek et al.'s implementation in Figure 2 with an instance of $\mathcal{B}$, and each Fetch&Add with an instance of $\mathcal{C}$. Let $\mathcal{A}$ be the resulting implementation. Clearly, $\mathcal{A}$ is an n-process wait-free linearizable implementation of a stack.

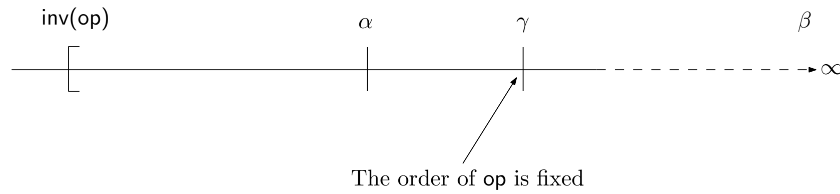

Figure 3 Universal helping: every pending operation is eventually linearized.

Moreover, it has no helping because, as mentioned already, $\mathcal{B}$ has no helping (in the sense described above) and Afek et al.'s stack implementation has no helping as well. Note that it does not matter if $\mathcal{C}$ has helping or not (trivial or nontrivial) because, as already pointed out, the only way a pop operation can become univalent is by winning the Test&Set in line 6, hence the $\mathcal{C}$ cannot change this. ◀

From Theorems 3 and 6, we get that nontrivial helping is a distinguishing factor between stacks and queues: while a stack can be implemented without nontrivial helping from read/write and Test&Set, any implementation of a queue from the same primitives necessitates nontrivial helping. Although the stack implementation of Theorem 6 is without *nontrivial* helping, it *does* have trivial helping. An example is when a process p reads the counter *range* in line 3 when there is only a single non-$\perp$ value in $T[1,\ldots,t]$ (where t is the value that p reads), and then a process $q \neq p$ reads *range* after p and takes the only non-$\perp$ value in $T[1,\ldots,t]$ (namely, q overtakes p). When q wins in line 6, it makes p's pop operation $\perp$-univalent because p will scan the range $T[1,\ldots,t]$ without seeing any non-$\perp$ value, and will therefore return $\perp$ in line 7.

4 Universal (Linearization-Based) Helping

In this section we propose another formalization of helping, in which a process ensures that operations by other processes are eventually linearized. This definition captures helping mechanisms such as the one used in Herlihy's universal wait-free construction [12]. We evaluate the power of this helping mechanism via consensus and compare it with the valency-based helping notion studied in Section 3.

Throughout this section, we assume, without loss of generality, that the first step of every operation is to publish its *signature* (i.e., the operation type and its parameters) to the shared-memory so that it may be helped by other processes. An operation is *pending* if it has published its signature but did not yet terminate.

▶ **Definition 7** (Universal (linearization-based) Helping)**.** Consider an n-process wait-free linearizable implementation of a data type T. The implementation has *universal helping* if every infinite extension $\alpha\beta$ of a finite history α has a finite prefix γ with a linearization $\mathsf{lin}(\gamma)$, which satisfies the following conditions:

- $\mathsf{lin}(\gamma)$ contains every pending high-level operation of α (see Figure 3), in addition to all high-level operations that complete in γ.
- Every extension $\gamma\lambda$ of γ has a linearization $\mathsf{lin}'(\gamma\lambda)$ such that $\mathsf{lin}'(\gamma)$ is a prefix of it.

If the above conditions are satisfied for every γ such that some process completes $f(n)$ or more high-level operations in the extension $\gamma - \alpha$, for some function $f : \mathbb{N} \to \mathbb{N}$, then we say the implementation has *f-universal helping*.

```
Operation propose(v_i):
(01)  Vals[i] ← v_i
(02)  Simulate to completion f(n) + 1 enq(i) operations
(03)  S ← Snapshot of the shared-memory variables used by B
(04)  d ← Locally simulate a deq() operation on B, starting from S
(05)  decide Vals[d]
endoperation
```

Figure 4 Solving consensus using $\mathcal{B}$.

Universal helping requires that the progress of some processes eventually ensures that all pending invocations are linearized and all processes make progress. f-universal helping bounds from above the number of high-level operations a process needs to perform in order to ensure the progress of other processes.

▶ **Theorem 8.** *Let $\mathcal{B}$ be an n-process nonblocking linearizable implementation of a queue or stack. If $\mathcal{B}$ has f-universal helping, then n-process consensus can be solved using $\mathcal{B}$ and read/write registers.*

Proof. First assume that $\mathcal{B}$ implements a queue. Figure 4 shows the pseudocode of an algorithm that solves consensus using $\mathcal{B}$ and read/write registers. Each process p_i first writes its proposal to $Vals[i]$ (initialized to $\perp$) in Line 01 and then performs $f(n)+1$ $\mathsf{enq}(i)$ operations in Line 02.

To solve consensus, p_i computes a snapshot that reads the state of the queue from the shared memory to a local variable S (Line 03) and then invokes a single $\mathsf{deq}()$ operation using state S in Line 04 (we say that p_i *locally simulates* the $\mathsf{deq}()$ operation). Finally, p_i decides (in Line 05) on the value proposed by the process whose identifier it dequeues in Line 04. The snapshot in Line 03 is taken as follows. In all executions of $\mathcal{B}$ in which each process executes at most $f(n)+1$ enqueue operations, processes access a finite set of base objects in the shared memory. Let R be the set with all base objects in all those executions. Then, processes use any read/write wait-free snapshot algorithm to take a snapshot of R. We now prove that the algorithm is correct.

Termination. Every correct process decides as each process invokes a finite number of operations of $\mathcal{B}$, which is nonblocking.

Validity. The view stored to S in Line 03 represents a state of $\mathcal{B}$ in which the queue is non-empty, since at least a single $\mathsf{enq}()$ operation completed and no $\mathsf{deq}()$ operations were invoked. Moreover, if p_i gets d from its local simulation, p_d participated in the execution. It follows that every correct process p_i decides on a proposed value.

Agreement. We prove that all correct processes dequeue the same value in Line 04, from which agreement follows easily. Let E be an execution of the algorithm of Figure 3. Let p_i, p_j be two distinct correct processes. Let α_i (resp. α_j) be the shortest prefix of E in which the first enqueue operation performed by p_i (resp. p_j) in Line 02 completes. Let γ_i (resp. γ_j) denote the shortest extension of α_i in which p_i (resp. p_j) completes the last enqueue operation in Line 02.

WLOG, assume that γ_i is a prefix of γ_j, that is, p_i is the first to complete Line 02. In γ_i, p_i completed $f(n)$ $\mathsf{enq}()$ operations after completing its first enqueue operation on $\mathcal{B}$. Consequently, Definition 7 guarantees that its first $\mathsf{enq}(i)$ operation, as well as any operations that preceded it and pending operations that were concurrent with it, are linearized in $lin(\gamma_i)$ and their order is fixed. Since no dequeue operations are applied to

(the shared copy of) B, $lin(\gamma_i)$ consists of enqueue operations only. Let $\mathsf{enq}(k)$ be the first operation in $lin(\gamma_i)$.
Let β_i (resp. β_j) denote the shortest prefix of E in which p_i (resp. p_j) completes Line 04. Since γ_i is a prefix of both β_i and β_j, it follows from Definition 7 that there are linearizations $lin'(\beta_i)$ and $lin'(\beta_j)$ in which $\mathsf{enq}(k)$ is the first operation. It follows in turn that the dequeue operations of both p_i and p_j in Line 04 return k, hence they both decide on $vals[k]$ in Line 05.

A similar argument gives the same result for stacks. The difference is that in the local simulation, a process simulates pop operations until it gets an *empty* response, and then decides on the proposed value of the process whose identifier was popped last (hence, pushed first). ◀

Herlihy's universal construction [12] has f-universal helping. Thus, Theorem 8 implies that, for stacks and queues, Herlihy's construction uses the full power of n-consensus in the sense that the resulting implementations can actually be used to solve n-consensus.

The next lemma will be used to show that for queues and stacks, universal helping implies nontrivial helping.

▶ **Lemma 9.** *Let T be a data type with two operations $\mathsf{put}(x)$ and $\mathsf{get}()$ such that, for distinct processes p and q, there is an infinite sequential execution S of T containing only put operations by q with a prefix S' such that:*

P1: *For every prefix $S' \cdot S''$ of S, in every sequential extension $S' \cdot S'' \cdot \langle p.\mathsf{get}() : \mathsf{return}\, y\rangle$, $y \neq \bot$ holds, where $\bot$ is the initial state of T.*

P2: *In every pair of sequential extensions $S' \cdot \langle p.\mathsf{get}() : \mathsf{return}\, y_1\rangle \cdot \langle q.\mathsf{get}() : \mathsf{return}\, z_1\rangle$ and $S' \cdot \langle q.\mathsf{get}() : \mathsf{return}\, z_2\rangle \cdot \langle p.\mathsf{get}() : \mathsf{return}\, y_2\rangle$, $y_1 \neq y_2$ holds.*

Then, any wait-free linearizable implementation of T with universal helping also has nontrivial helping.

Proof. Consider sequential executions S and S' of T as the lemma assumes. Let $\mathcal{A}$ be a wait-free linearizable implementation of T with universal helping. Let α be an execution of $\mathcal{A}$ in which q completes alone all operations in S', in that order. We claim that a $\mathsf{get}_p()$ by p is multivalent in the configuration at the end of α (note that at the end of α, $\mathsf{get}_p()$ has not even started): by property P2, the output of $\mathsf{get}_p()$ in the extension of α in which $\mathsf{get}_p()$ is completed alone and then a $\mathsf{get}_q()$ by q is completed alone, is different from the output in which the operations are completed in the opposite order.

Now, let α' be an extension of α in which p executes alone a $\mathsf{get}_p()$ until the operation is critical. Let β be an infinite extension of α' in which q runs alone and executes the operations in $S - S'$, in that order. Since $\mathcal{A}$ has universal helping, there is a finite prefix γ of β such that there is a linearization $\mathsf{lin}(\gamma)$ containing $\mathsf{get}_p()$. Moreover, for every extension λ of γ, there is a linearization $\mathsf{lin}'(\lambda)$ such that $\mathsf{lin}(\gamma) = \mathsf{lin}'(\gamma)$. Intuitively, this means that the linearization order of $\mathsf{get}_p()$ in γ is fixed, hence it is univalent at the end of γ. We formally prove this.

Let v be the return value of $\mathsf{get}_p()$ in $\mathsf{lin}(\gamma)$. We claim that $\mathsf{get}_p()$ is v-univalent in the configuration at the end of γ. Let λ be any extension of γ in which $\mathsf{get}_p()$ is completed. Let u be the output value of $\mathsf{get}_p()$ in λ. Since $\mathsf{get}_p()$ is completed in λ, any linearization of λ contains $\mathsf{get}_p()$. As noted above, there is a linearization $\mathsf{lin}'(\lambda)$ of λ such that $\mathsf{lin}(\gamma) = \mathsf{lin}'(\gamma)$, which implies that $u = v$. We conclude that $\mathsf{get}_p()$ is v-univalent at the end of γ. We now show that $v \neq \bot$. Observe that $\mathsf{lin}(\gamma)$ must have the form $S' \cdot S'' \cdot \langle \mathsf{get}_p() : \mathsf{return}\, v\rangle \cdot S'''$, for some sequences S'' and S''' of put operations by q. Note that $S' \cdot S'' \cdot S'''$ is a prefix of

S, since q executes its operations in the order they appear in S. Thus, by property P1, it follows that $v \neq \bot$.

Finally, since $\mathsf{get}_p()$ is multivalent at the end of α' and it is univalent on a non-$\bot$ value at the end of γ, there must be a prefix of γ that ends in a configuration C in which $\mathsf{get}_p()$ is multivalent but it is univalent in $q(C)$ on a non-$\bot$ value. Therefore, $\mathcal{A}$ has nontrivial helping. ◀

▶ **Corollary 10.** *A wait-free linearizable implementation of a queue or stack with universal helping has nontrivial helping.*

Proof. For the case of the stack, S is the infinite execution in which some process q performs a sequence of *push* operations with distinct values and S' is any non-empty prefix of S. The case of the queue is defined similarly. ◀

Figure 5 presents a stack implementation that has nontrivial helping but not universal helping, as established by Lemma 14 in the appendix. It augments the wait-free stack of Afek et al. [1] with a helping mechanism, added by lines 01–05 in push and lines 08–14 in pop. Each process $p_i \neq p_n$ that wants to push value x, first checks if p_n's current pop operation is pending (lines 02–03), and if so, tries to help p_n by directly giving x to its current pop operation. If p_i succeeds in updating $H[j]$ in line 05, then it does not access the *items* array. In that case, p_n is not able to update $H[j]$ in line 11, implying that it must take the value in *h_items* that p_i left for it (lines 12–14). If p_n manages to update $H[j]$ in line 11, then no process succeeded in helping it and it proceeds as in Afek et al.'s stack (lines 15–23). Similarly, processes whose *Push* operation fails to help p_n proceed as in Afek et al.'s stack (lines 06–07).

In Appendix A, we prove that the algorithm in Figure 5 is a wait-free linearizable implementation of a stack that has nontrivial helping but not universal helping. Together with Lemma 9, this implies that, for stacks, universal helping is strictly stronger than nontrivial helping.

5 Related Notions

This section compares valency-based helping and universal helping to the definition of helping in [3] and the notion of strong linearizability [9].

5.1 Relation to the help definition of [3]

Helping is formalized in [3] as follows. A linearizable implementation of a concurrent object has helping, which we call here *linearization-based helping*, if there is an execution α with distinct operations op_1 and op_2 by p and q, such that

1. there are linearizations $\mathsf{lin}(\alpha)$ and $\mathsf{lin}'(\alpha)$ such that op_1 precedes op_2 in $\mathsf{lin}(\alpha)$ and op_2 precedes op_1 in $\mathsf{lin}'(\alpha)$, and
2. in every linearization of $\alpha \cdot r$, for some $r \neq p$ (possibly $r = q$), op_1 precedes op_2.

In a sense, valency-based helping and linearization-based helping are incomparable. On the one hand, valency-based helping allows us to distinguish stacks and queues, as queues need nontrivial (valency-based) helping and stacks do not (Theorems 3 and 6), while both stacks and queues necessarily have linearization-based helping [3]. On the other hand, valency-based helping cannot capture helping among enqueues, as they always return true. Nevertheless, enqueues *are* taken into account, since the dequeues in an execution reveal how the helping mechanism determines the order of enqueues.

```
Shared Variables:
    range : Fetch&Add register initialized to 1
    current : read/write register initialized to 1
    items : array [1,...,] of read/write registers
    h_items : 2-dimensional array [1,...,n−1,1,...,] of read/write registers
    pend : array [1,...,] of boolean read/write registers initialized to false
    T : array [1,...,] of Test&Set objects
    H : array [1,...,] of Compare&Swap objects initialized to ⊥

Operation Push(x):
(01)  if ID ≠ n then
(02)      j ← current
(03)      if pend[j] then
(04)          h_items[ID][j] ← x
(05)          if Compare&Swap(H[j], ⊥, ID) then return  true
          end if
      end if
(06)  i = Fetch&Add(range, 1)
(07)  items[i] ← x
      return  true
end Push

Operation Pop():
(08)  if ID = n then
(09)      j ← current
(10)      pend[j] ← true
(11)      if ¬Compare&Swap(H[j], ⊥, ID) then
(12)          current ← j + 1
(13)          k ← get(H[j])
(14)          return h_items[k][j]
          end if
      end if
(15)  t = Fetch&Add(range, 0)
(16)  for i ← t downto 1 do
(17)      x ← items[i]
(18)      if x ≠ ⊥ then
(19)          if Test&Set(T[i]) then
(20)              if ID = n then current ← j + 1
(21)              return x
              end if
          end if
      end for
(22)  if ID = n then current ← j + 1
(23)  return ⊥
end Pop
```

Figure 5 A stack implementation without universal helping.

A proof similar to the proof of Lemma 9 shows that universal helping implies linearization-based helping. Consider any implementation (wait-free or nonblocking) of a data type with universal helping. Consider an infinite execution β that starts in an initial configuration, where a process p starts some operation op_1 and stops before the operation is completed, and afterwards a distinct process q completes infinitely many operations (the type does not matter). Since the implementation has universal helping, eventually, the order of op_1 in any linearization is fixed. More precisely, there is a prefix γ of β such that, for every linearization of every extension of γ, the order of op_1 is the same. This implies that, at some point, a step of q made op_1 precede an operation op_2 of q, in every linearization. Thus, the implementation has linearization-based helping.

The other direction is not necessarily true, since one can modify Herlihy's universal construction to get a nonblocking implementation of any data type from Compare&Swap in

which each process is helped just once. The resulting implementation has linearization-based helping but not universal helping because universal helping requires that *every* pending operation is eventually linearized, which does not happen once every process in the execution has been helped, since from this point on some operations may be blocked forever.

▶ **Theorem 11.** *For every data type T, every nonblocking or wait-free implementation of T with universal helping has linearization-based helping, while the opposite is not necessarily true.*

5.2 Relation to strong linearizability [9]

Roughly speaking, an implementation of a data type is *strongly linearizable* [3] if once an operation is linearized, its linearization order cannot be changed in the future. More specifically, there is a function L mapping each execution to a linearization, and the function is *prefix-closed*: for every two executions α and β, if α is a prefix of β, then $L(\alpha)$ is a prefix of $L(\beta)$.

In a sense, universal helping can be thought of as a sort of *eventual* strong linearizability. For every execution α, as it is extended, there is eventually an extension α' with a linearization $\mathsf{lin}(\alpha\,\alpha')$ such that for every execution β, if $\alpha\,\alpha'$ is a prefix of β, then there is a linearization $\mathsf{lin}'(\beta)$ with $\mathsf{lin}(\alpha\,\alpha') = \mathsf{lin}'(\alpha\,\alpha')$. We stress that universal helping provides the property that pending operations are linearized eventually, which is not guaranteed by strong linearizability.

The simulation in the proof of Theorem 8 solves consensus because from some point on, all processes agree on a first operation and this agreement cannot be changed as a result of future steps. The following theorem can be proven using a simulation similar to the one in the proof of Theorem 8, with the difference being that each process only needs to complete a single enqueue because the linearization order of that operation does not change in the future.

▶ **Theorem 12.** *Let $\mathcal{B}$ be an n-process strongly-linearizable nonblocking implementation of a queue (stack). Then, n-process consensus can be solved from $\mathcal{B}$.*

The previous theorem shows that, for some data types, strong linearizability for n processes can only be obtained through consensus number n, thus strong linearizability is costly, even if we are looking for nonblocking implementations. However, for stacks, linearizability can be obtained from consensus number 2 as there are wait-free stack implementations from Test&Set [1].

▶ **Corollary 13.** *There is no n-process strongly-linearizable nonblocking implementation of a queue (stack) from primitives with consensus number less than n.*

All previous impossibility results on strongly-linearizable implementations that we are aware of consider only implemenations from consensus-number 1 base objects [7, 10].

6 Discussion

We have considered two ways to formalize helping in implementations of shared objects, one that is based on operation valency and another that is based on possible linearizations. We used these notions to study the kind of helping needed in wait-free implementations of queues and stacks, from Test&Set and stronger primitives. In this work we used an ad-hoc definition of nontrivial helping for queues and stacks, but this notion can be generalized by defining two disjoint sets of outputs values, *trivial* and *nontrivial*, and defining trivial and

nontrivial helping accordingly. These notions might facilitate further study of the relations between nonblocking and wait-free implementations.

References

1 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 20(4):239–252, 2007.

2 Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC'93, pages 159–170, 1993.

3 Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC'15, pages 241–250, 2015.

4 Matei David. A single-enqueuer wait-free queue implementation. In *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, pages 132–143, 2004. doi:10.1007/978-3-540-30186-8_10.

5 Matei David. Wait-free linearizable queue implementation. Master's thesis, Department of Computer Science, University of Toronto, 2004.

6 Matei David, Alex Brodsky, and Faith Ellen Fich. Restricted stack implementations. In *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, pages 137–151, 2005. doi:10.1007/11561927_12.

7 Oksana Denysyuk and Philipp Woelfel. Wait-freedom is harder than lock-freedom under strong linearizability. In *Distributed Computing – 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 60–74, 2015. doi:10.1007/978-3-662-48653-5_5.

8 David Eisenstat. A two-enqueuer queue. *CoRR*, abs/0805.0444, 2008. URL: http://arxiv.org/abs/0805.0444.

9 Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, STOC'11, pages 373–382, 2011.

10 Maryam Helmi, Lisa Higham, and Philipp Woelfel. Strongly linearizable implementations: possibilities and impossibilities. In *ACM Symposium on Principles of Distributed Computing, PODC'12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 385–394, 2012. doi:10.1145/2332432.2332508.

11 Danny Hendler and Nir Shavit. Operation-valency and the cost of coordination. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC'03, pages 84–91, 2003.

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

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

14 Zongpeng Li. Non-blocking implementations of queues in asynchronous distributed shared-memory systems. Master's thesis, Department of Computer Science, University of Toronto, 2001.

A Stack without Universal Helping

▶ **Lemma 14.** *The algorithm in Figure 5 is a wait-free linearizable implementation of a stack that has nontrivial helping but no universal helping.*

Proof. We first prove that the implementation is linearizable (clearly, it is wait-free). Let α be an execution of the algorithm. Intuitively, we show that there is an execution γ of Afek et al.'s stack implementation (see Figure 2) such that the operations in γ respect the real-time order of the operations in α and the outputs are the same. Thus, α is linearizable since γ is linearizable.

In any execution of the algorithm, a $\mathsf{push}_i(x)$ operation of p_i *matches* the j-th $\mathsf{pop}_n()$ operation of p_n, if p_i successfully updates $H[j]$ during the execution. We call such a pair of operations a *matching*.

Let k be the number of matchings in α. By induction on k, we show that α is linearizable. If $k = 0$, then α is linearizable because it corresponds to some execution of Afek et al.'s implementation. Suppose that the claim holds for $k - 1$. Below we show that it holds for k.

Let $\mathsf{push}_i(x)$ by p_i and $\mathsf{pop}_n()$ by p_n be the k'th matching in α. Note that $\mathsf{push}_i(x)$ and $\mathsf{pop}_n()$ are concurrent in α. Moreover, the Compare&Swap in line 05 of $\mathsf{push}_i(x)$ precedes the Compare&Swap in line 11 of $\mathsf{pop}_j()$ (if p_n ever executes it).

We now construct an execution α' that is easier to reason about than α. Let β be the longest prefix of α that does not have the Compare&Swap in line 05 of $\mathsf{push}_i(x)$. Thus, in the configuration at the end of β, p_i is about to perform the Compare&Swap in line 05, and p_n is about to perform the Compare&Swap in line 11.

Let β_i and β_n respectively denote the subsequences of $\alpha - \beta$ containing only the steps of $\mathsf{push}_i(x)$ and $\mathsf{pop}_n()$. Let λ be the subsequence of $\alpha - \beta$ obtained by removing the steps of β_i and β_n.

Then, α' is the execution $\beta\,\beta_i\,\beta_n\,\lambda$. Intuitively, in α', the steps of $\mathsf{push}_i(x)$ and $\mathsf{pop}_n()$ are placed together.

It can be seen that there is no process that can distinguish between α and α': since neither p_i nor p_n change *items* or *range* in $\mathsf{push}_i(x)$ and $\mathsf{pop}_n()$, the position in the execution when they take the steps in β_i and β_n does not affect other operations. Moreover, α' respects the real-time order in α: if an operation op_1 precedes op_2 in α, op_1 also precedes op_2 in α'. Although there may be concurrent operations in α that are not concurrent in α', this is not a problem for linearizability.[1] Therefore, if α' is linearizable, then α is linearizable too. We now show that it is.

Consider the following execution γ that starts with β and then:

1. p_n executes the Compare&Swap in line 11 (hence sets $H[j]$).
2. p_i executes three consecutive steps, which correspond to lines 05, 06 and 07 (because it cannot set $H[j]$).
3. If $\mathsf{pop}_j()$ (by p_n) is completed in α', p_n completes it in γ (thus it outputs x).
4. If $\mathsf{push}_i(x)$ is completed in α', p_i completes it in γ.
5. λ is appended at the end.

Thus, in γ, p_n is about to take its output from *items*, p_i places x in *items* (at the top of the stack) and p_n takes it from there. The steps of λ, following $\mathsf{push}_i(x)$ and $\mathsf{pop}_j()$, proceed as in α and the only difference is that the Fetch&Add in lines 06 and 15 outputs in γ an integer larger than in α, since p_i adds 1 to *range* in γ in operation $\mathsf{push}_i(x)$.

Also observe that γ respects the real-time order in α'. By induction hypothesis, γ is linearizable, since it has $k - 1$ matchings. Let $\mathsf{lin}(\gamma)$ be a linearization of γ. From the

[1] For example, in α', $\mathsf{push}_i(x)$ precedes any operation starting in λ, however, in α those operations might be concurrent.

properties of γ just described, it follows that $\mathsf{lin}(\gamma)$ is actually a linearization of α' as well, hence a linearization of α. Therefore, the implementation is linearizable.

We now show that the algorithm has nontrivial helping. Starting at the initial configuration, let α be the execution in which p_n completes alone a $\mathsf{push}(1)$ operation and then starts a $\mathsf{pop}()$ operation and stops just before executing the Compare&Swap in line 11. Then, p_1 starts a $\mathsf{push}(2)$ operation and stops just before executing the Compare&Swap in line 05.

Let C be the configuration at the end of α. We claim that the $\mathsf{pop}()$ is multivalent in C. Indeed, let $x \geq 3$. In the extension of α in which first p_2 completes a $\mathsf{push}(x)$ alone and then p_n completes its $\mathsf{pop}()$, the output of the $\mathsf{pop}()$ is x. Also, note that $\mathsf{pop}()$ is 2-univalent in $p_1(C)$ because there is no extension of $p_1(C)$ in which p_n updates $H[1]$ in Line 11, so if it ever returns a value, this must be the value in $h_items[1][1]$ (which is 2).

Finally, we prove that the implementation has no universal helping. Starting at the initial configuration, let α be the execution in which p_1 starts $\mathsf{pop}()$ and stops before executing the Fetch&Add in line 15. Let β the the infinite extension of α in which p_2 completes alone (infinitely many) push operations with distinct values. If the algorithm would have had universal helping, then there would have been a finite prefix γ of β such that there was a linearization $\mathsf{lin}(\gamma)$ containing pop(), and for every extension λ of γ, there would have been a linearization $\mathsf{lin}'(\lambda)$ such that $\mathsf{lin}(\gamma) = \mathsf{lin}'(\gamma)$.

Let γ be such a prefix of β and let λ be the extension of γ in which p_2 completes any pending operation in γ and a $\mathsf{push}(x)$, where x is greater than any value in γ. Let λ' be the extension of λ in which p_1 completes its pop() operation. Observe that p_1's operation outputs x in λ'. Moreover, there is no linearization $\mathsf{lin}'(\lambda')$ of λ' with $\mathsf{lin}(\gamma) = \mathsf{lin}'(\gamma)$ because $\mathsf{push}(x)$ does not appear in γ. Thus, the implementation has no universal helping. ◀