On Helping and Stacks

Vitaly Aksenov[1,2], Petr Kuznetsov[3], and Anatoly Shalyto[1]

[1] ITMO University, Russia
[2] Inria Paris, France
[3] LTCI, Télécom ParisTech, Université Paris-Saclay

Abstract. A concurrent algorithm exhibits *helping* when one process performs work on behalf of other processes. More formally, helping is observed when the order of some operation in a linearization is *fixed* by a step of another process. In this paper, we show that no wait-free linearizable implementation of a *stack* using read, write, compare&swap and fetch&add operations can be *help-free*, correcting a mistake in an earlier proof by Censor-Hillel et al.

1 Introduction

In a *wait-free* data structure, every process is guaranteed to make progress in its own speed, regardless of the behavior of other processes [8]. It has been observed, however, that achieving wait-freedom typically involves some *helping* mechanism (e.g., [6,14,7,13]). Informally, helping means that a process may perform additional work on behalf of other processes.

Censor-Hillel et al. [5] proposed a natural formalization of the concept of helping, based on the notion of *linearization*: a process p helps an operation of a process q in a given execution if a step of p determines that an operation of q takes effect, or *linearizes*, before some other operation in any possible extension. It was claimed in [5] that helping is required for any wait-free linearizable implementation of an *exact order* data type in a system provided with read, write, compare&swap and fetch&add shared memory primitives. Informally, a sequential data type is exact order if for some operation sequence, every change in the relative order of two operations affects the result of some other operations. As examples of exact order data types, Censor-Hillel et al. gave (FIFO) queue and (LIFO) stack.

We observe, however, that the stack data type is not exact order. As we show, in any sequential execution on stack, we can reorder any two operations *op* and *op′* in such a way that no other operation will see the difference. Hence, the proof of help-free impossibility for exact order types given in [5] does not apply to stack.

In this paper, we propose a direct proof that stack does not have help-free implementations. At first, we show the result for implementations using read, write and compare&swap operations in systems with at least three processes, and, then, extend the proof to those additionally using fetch&add in systems with at least four proccesses. The structure of the proofs generally follows the structure from the paper by Censor-Hillel et al. [5], but the underlying reasoning is novel. Unlike their approach our proofs argue about the order of operations given their responses *only* after we empty the data structure. As a result, certain steps of the proof become more technically involved.

The paper is organized as follows. In Section 2 we present a computational model and necessary definitions. In Section 3 we recall the definition of helping and highlight the mistake in [5]. In Section 4 we give our direct proof. In Section 5 we discuss the related work. And, finally, we conclude in Section 6.

2 Model and definitions

We consider a system of n processes $p_1, \ldots, p_n$ communicating via invocations of *primitives* on a shared memory. We assume that primitives are *read*, *write* and *compare&swap*. In our second technical contribution, we consider one more primitive *fetch&add*.

A compare&swap primitive takes a target location, an expected value and a new value. The value stored in the location is compared to the expected value. If they are equal, then the value in the location is replaced with the new value and `true` is returned (we say that the operation is *successful*). Otherwise, the operation *fails* (i.e., the operation is *failed*) and returns `false`.

A fetch&add primitive takes a target location and an integer value. The primitive augments the value in the location by the provided value and returns the original value.

A high-level concurrent object or a data type is a tuple $(\Phi, \Gamma, Q, q_0, \theta)$, where Φ is a set of operations, Γ is a set of responses, Q is a seq of states, q_0 is an initial state and a transition function $\theta \subset Q \times \Phi \times Q \times \Gamma$, that determines, for each state and each operation, the set of possible resulting states and produced responses.

In this paper, we concentrate on a stack data type (further, we omit "data type" and simply refer to it as "stack"). It exports two methods `push`$(\cdot)$ and `pop`$()$. A `push`(x) operation pushes an element at the top of the stack. A `pop`$()$ operations withdraws and returns the element from the top of the stack, or returns $\perp$, if the stack is empty.

An implementation (or, simple object) of a high-level object O is a distributed algorithm A consisting of local state machines $A_1, \ldots, A_n$. A_i specifies the primitives p_i needs to execute to return a response to an invoked operation on O. For simplicity, all implementations considered in this paper are deterministic. Nevertheless, as we pursue impossibility results, the proofs easily extend to randomized implementations. For the rest of the section we fix some implementation of stack.

A *program* of a process specifies a sequence of operations calls on an object. The program may include local computations and can choose which operation to execute depending on the results of the previous operations.

A *history* is a finite or infinite sequence of primitive steps. Each step is coupled with a specific operation that is being executed by the process performing this step. The first step of an operation always comes with the input parameters of the operation, and the last step of an operation is associated with the return of the operation. Given two histories h_1 and h_2 we denote by $h_1 \circ h_2$ the concatenation of h_1 and h_2.

A *schedule* is a finite or infinite sequence of process ids. Given a schedule, an implementation and programs provided to the processes, one can unambiguously determine the corresponding history. And vice versa, given a history one can always build a schedule by substituting the steps of history to the process that performed it. Assuming a fixed program for each process (these programs will be clear from the context), and a history h, we denote by $h \circ p_i$ the history derived from scheduling process p_i to take the next step (if any) following its program immediately after h.

The set of histories H induced by an implementation consists of all possible histories induced by all possible processes' programs with all possible schedules. Note that, by the definition, H is prefix- and limit-closed [10].

A history defines a partial order on the operations: op_1 precedes op_2 in a history h (denoted: $op_1 \prec_h op_2$) if op_1 is completed before op_2 begins. A *linearization* L of a history h is a sequence of operations such that 1) L consists of all the completed operations and, possibly, some started but incompleted in h; 2) the operations have the same input and same output as corresponding operations in h; 3) L consistent with the data type; 4) for every two operations $op_1 \prec_h op_2$ if op_2 is included in L, then op_1 preceds op_2 in L ($op_1 \prec_L op_2$).

An implementation of a data type is linearizable if each history from the set of histories has a linearization. A *linearization function* defined over a set of linearizable histories H maps every history in H to a linearization. Note that a linearizable implementations may have multiple linearization functions defined on the set of its histories.

An implementation is *wait-free* if every process completes its operation in a finite number of steps.

3 Helping and Exact Order Types

In this section, we recall the definitions of helping and exact order type in [5] and show that stack is *not* exact order.

Definition 1 (Decided before). *For a history h in a set of histories H, a linearization function f over H, and two operations op_1 and op_2, we say that op_1* is decided before *op_2 in h with respect to f and H, if there exists no extension $s \in H$ of h such that $op_2 \prec_{f(s)} op_1$.*

Definition 2 (Helping). *A set of histories H with a linearization function f over H is* help-free *if for every $h \in H$, every two operations op_1, op_2, and a single computation step γ such that $h \circ \gamma \in H$ it holds that if op_1 is decided before op_2 in $h \circ \gamma$ and op_1 is not decided before op_2 in h then γ is a step in the execution of op_1.*

An implementation is help-free*, if there exists a linearization function f such that the set of histories of this implementation with f is help-free.*

Following the formalism of [5], if S is a sequence of operations, we denote by $S(n)$ the first n operations in S, and by S_n the n-th operation of S. We denote by $(S + op?)$ the set of sequences that contains S and all sequences that are similar to S, except that a single operation op is inserted somewhere between (or before, or after) the operations of S.

Definition 3 (Exact Order Types). *An* exact order type *is a data type for which there exists an operation op, an infinite sequence of operations W, and a (finite or an infinite) sequence of operations R, such that for every integer $n \geq 0$ there exists an integer $m \geq 1$, such that for any sequence A from $W(n+1) \circ (R(m) + op?)$ and any sequence B from $W(n) \circ op \circ (R(m) + W_{n+1}?)$ at least one operation in $R(m)$ has different results in A and B, where $\circ$ is a concatenation of sequences.*

It is shown in [5] that exact order types require helping, when implemented with read, writes, and compare&swap primitives. The paper also sketches the proof of a more general result for systems that, additionally, use fetch&add. Further, it is claimed in [5] that stack and queue are exact order types. Indeed, at first glance, if you swap two subsequent operations, further operations have to acknowledge this difference. However, the definition of an exact order type is slightly more complicated, as it allows not only to swap operations but also move them. This relaxation does not affect queue, but, unfortunately, it affects stack.

Theorem 1. *Stack is not an exact order type.*

Proof. We prove that for any fixed op, W, R and n there does not exist m that satisfies Definition 3. Note that the claim is stronger than what is needed to prove the theorem: it would be sufficient to prove that for all op, W and R, the condition does not hold *for some n*. In a sense, this suggests that stack is *far* from being exact order.

Suppose, by contradiction, that there exists m that satisfies Definition 3 for fixed op, W, R and n. There are four cases for op and W_{n+1}: `pop-pop`, `push-pop`, `pop-push` or `push-push`. For each of these cases, we find two sequences from $W(n+1) \circ (R(m) + op?)$ and $W(n) \circ op \circ (R(m) + W_{n+1}?)$ for which all operations in $R(m)$ return the same results.

- $op = \texttt{pop}$, $W_{n+1} = \texttt{pop}$. Then, $W(n+1) \circ op \circ R(m)$ and $W(n) \circ op \circ W_{n+1} \circ R(m)$ satisfy, since $W_{n+1} \circ op$ and $op \circ W_{n+1}$ perform two `pop` operations.
- $op = \texttt{push}(a)$, $W_{n+1} = \texttt{pop}$. For the first sequence we take $A = W(n+1) \circ op \circ R(m)$. Now, we choose the second sequence B from $W(n) \circ op \circ (R(m) + W_{n+1}?)$. Let W_{n+1} pop in A the x-th element from the bottom of the stack. We extend $W(n) \circ op$ in B with operations from $R(m)$ until some operation op' tries to pop the x-th element from the bottom. Note that all operations $R(m)$ up to op' (not including op') return the same results in A and B. If such op' does not exist then we are done. Otherwise, we insert W_{n+1} right before op', i.e., pop this element. Subsequent operations in $R(m)$ are not affected, i.e., results of operations in $R(m)$ are the same in A and B.
- $op = \texttt{pop}$, $W_{n+1} = \texttt{push}(b)$. This case is symmetric to the previous one.

- $op = \texttt{push}(a)$, $W_{n+1} = \texttt{push}(b)$. For the first sequence, we take $A = W(n+1) \circ op \circ R(m)$. Now, we build the second sequence B from $W(n) \circ op \circ (R(m) + W_{n+1}?)$. Let W_{n+1} push in A the x-th element from the bottom of the stack. Let us perform $W(n) \circ op$ in B and start performing operations from $R(m)$ until some operation op' pops the x-th element (again, this should eventually happen, otherwise a contradiction is established). Note that all operations $R(m)$ up to op' (including op') return the same results in A and B. If such op' does not exist then we are done. Otherwise, right after op' we perform W_{n+1}, i.e., push the element b in its proper position. Subsequent operations in $R(m)$ are not affected and, thus, the results of all operations in $R(m)$ are the same in A and B.

The contradiction implies that stack is not an exact order type.

4 Wait-free stack cannot be help-free

In this section, we prove that there does not exist a help-free wait-free implementation of stack in a system with reads, writes, and compare&swaps. We then extend the proof to the case when a system has one more primitive fetch&add.

4.1 Help-free stacks using reads, writes and compare&swap

Suppose that there exists such a help-free stack implementation Q using read, write, and compare&swap primitives. We establish a contradiction by presenting a history h in which some operation takes infinitely many steps without completing.

We start with three observations that immediately follow from the definition of linearizability.

Observation 1 *In any history h:*

1. *Once an operation is completed it must be decided before all operations that have not yet started;*
2. *If an operation is not started it cannot be decided before any operation of a different process.*

Lemma 1 (Transitivity). *For any linearization function f and finite history h, if an operation op_2 is completed in h, an operation op_1 is decided before op_2 in h and op_2 is decided before an operation op_3 in h then op_1 is decided before op_3 in h.*

Proof. Suppose that op_1 is not decided before op_3 in h then there exists a extension s of h for which $op_3 \prec_{f(s)} op_1$. Since op_2 is linearized in $f(s)$ and op_1 is decided before op_2 then $op_1 \prec_{f(s)} op_2$. Together, $op_3 \prec_{f(s)} op_1 \prec_{f(s)} op_2$ contradicting with op_2 being decided before op_3 in h.

Lemma 2. *For any linearization function f and finite history h, if an operation op_1 of a process p_1 is decided before an operation op_2 of a process p_2, then op_1 must be decided before any operation op that has not started in h.*

Proof. Consider h', the extension of h, in which p_2 runs solo until op_2 completes. Such an extension exists, as Q is wait-free. By Observation 1 (1), op_2 is decided before op in h', and, consequently, by Transitivity Lemma 1, op_1 is decided before op in h'.

Since in h', only p_2 takes steps starting from h, op_1 must be decided before op in h — otherwise, h' has a prefix h'' such that op_1 is not decided before op in h'' and op_1 is decided before op in $h'' \circ p_2$ — a contradiction with the assumption that Q is help-free.

Now we build an *infinite* history h in which p_1 executes infinitely many failed compare&swap steps, yet it never completes its operation. We assume that p_1, p_2 and p_3 are assigned the following programs: p_1 tries to perform $op_1 = \texttt{push}(1)$; p_2 applies an infinite sequence of operations $\texttt{push}(2), \texttt{push}(3), \texttt{push}(4), \ldots$; and p_3 is about to perform an infinite sequence of $\texttt{pop}()$ operations.

The algorithm for constructing this history is given in Listing 1.1. Initially, p_1 invokes $op_1 = \texttt{push}(1)$ and, concurrently, p_2 invokes $op_2 = \texttt{push}(2)$. Then we interleave steps of p_1 and p_2 until a *critical* history h is located: op_1 is decided before op_2 in $h \circ p_1$ and op_2 is decided before op_1 in $h \circ p_2$. We let p_2 and p_1 take the next step and, then, run op_2 after $h \circ p_2 \circ p_1$ until it completes. We will show that op_1 cannot complete and that we can reiterate the construction by allowing p_2 to invoke concurrent operations $\texttt{push}(3)$, $\texttt{push}(4)$, etc. In the resulting infinite history, p_1 takes infinitely many steps without completing op_1.

```
1  h ← ϵ
2  op_1 ← push(1)
3  id_2 ← 2
4  while true:                // outer loop
5    op_2 ← push(id_2)
6    while true:              // inner loop
7      if op_1 is not decided before op_2 in h ∘ p_1:
8        h ← h ∘ p_1
9        continue
10     if op_2 is not decided before op_1 in h ∘ p_2:
11       h ← h ∘ p_2
12       continue
13     break
14   h ← h ∘ p_2
15   h ← h ∘ p_1
16   while op_2 is not completed
17     h ← h ∘ p_2
18   id_2 ← id_2 + 1
```

Listing 1.1: Constructing the history for the proof of Theorem 2

To ensure that at each iteration op_1 is not completed, we show that, at the start of each iteration of the outer loop (Line 5), the constructed history satisfies the following two invariants:

- op_1 is not decided before op_2 or before any operation of p_3;
- the operations of p_2 prior to op_2 are decided before op_1.

At the first iteration, the invariants trivially hold, since neither op_1 nor op_2 is started.

Observation 2 *The order between op_1 and op_2 cannot be decided during (and right after) the inner loop (Lines 6-13).*

Observation 3 *Process p_3 never takes a step in h.*

Lemma 3. *During (and right after) the execution of the inner loop (Lines 6-13) op_1 and op_2 cannot be decided before any operation of p_3.*

Proof. Suppose that during an execution of the inner loop op_1 or op_2 is decided before some operation of p_3.

Before entering the inner loop, neither op_1 nor op_2 is decided before any operation of p_3: op_1 is not decided because of the first invariant, while op_2 is not started (Observation 1 (2)). Thus, at least one step is performed by p_1 or p_2 during the execution of the inner loop.

Let us execute the inner loop until the first point in time when op_1 or op_2 is decided before an operation of p_3. Let this history be h. Note, that because Q is help-free only one of op_1 and op_2 is decided before an operation of p_3 in h. Suppose, that op_1 is decided before some op_3 of p_3, while op_2 is not decided before any operation of p_3. (The case when op_2 is decided before some op_3 is symmetric)

Now, p_3 runs `pop` operations until it completes operation op_3 and then, further, until the first `pop` operation returns $\perp$, i.e., the stack gets empty. Let the resulting extension of h be h'.

Recall that op_2 is not decided before any operation of p_3 in h and, since Q is help-free and only p_3 takes steps after h, op_2 cannot be decided before any operation of p_3 in h'. Hence, none of the completed operations of p_3 can return id_2, the argument of op_2 due to the fact that all `push` operations have different arguments. Because the operations of p_3 empty the stack op_2 has to linearize after them, making op_3 to be decided before op_2 in h'. By Transitivity Lemma 1, op_1 is decided before op_2 in h'. Finally, since Q is help-free and only p_3 takes steps after h op_1 has to be decided before op_2 in h, contradicting Observation 2.

Lemma 4. *op_1 and op_2 cannot be completed after the inner loop (Lines 6-13).*

Proof. Suppose the contrary. By Observation 1 (1), op_1 has to be decided before all operations of p_3, contradicting Lemma 3.

Lemma 5. *The execution of the inner loop (Lines 6-13) is finite.*

Proof. Suppose that the execution is infinite. By Lemma 4, neither of op_1 and op_2 is completed in h. Thus, in our infinite execution either op_1 or op_2 takes infinite number of steps, contradicting wait-freedom of Q.

Lemma 6. *Just before Line 14 the following holds:*

1. *The next primitive step by p_1 and p_2 is to the same memory location.*

2. The next primitive step by p_1 and p_2 is a compare&swap.
3. The expected value of the compare&swap steps of p_1 and p_2 is the value that appears in the designated address.
4. The new values of the compare&swap steps of p_1 and p_2 are different from the expected value.

Proof. Suppose that the next primitive steps by p_1 and p_2 are to different locations. Consider two histories: $h' = h \circ p_1 \circ p_2 \circ$ complete $op_1 \circ$ complete op_2 and $h'' = h \circ p_2 \circ p_1 \circ$ complete $op_1 \circ$ complete op_2. Let us look at the first two `pop()` operations by p_3. Executed after h' they have to return id_2 then 1, since op_1 is decided before op_2 in h' and both of them are completed. While executed after h'' they have to return 1 then id_2. But the local states of p_3 and shared memory states after h' and h'' are identical and, thus, two `pops` of p_3 must return the same values — a contradiction. The same argument will apply when both steps by p_1 and p_2 are reads.

Suppose that the next operation of p_1 is a write. (The case when the next operation of p_2 is write is symmetric) Consider two histories: $h' = h \circ p_2 \circ p_1 \circ$ complete op_1 and $h'' = h \circ p_1 \circ$ complete op_1. Let the process p_1 perform two `pop()` operations (op'_1 and op''_1) and p_2 complete its operation after h': op'_1 and op''_1 have to return 1 and id_2, correspondingly, since op_1 and op_2 are completed and op_2 is decided before op_1 in h'. Again, since the local states of p_1 and the shared memory states after h' and h'' are identical, op'_1 and op''_1 performed by p_1 after h'' must return 1 and id_2. Hence, op_2 has to be decided before op''_1 in $\tilde{h} = h'' \circ$ perform $op'_1 \circ$ perform op''_1 and, by Lemma 2, op_2 has to be decided before any operation of p_3 in $\tilde{h}$. Since only p_1 performs steps after h in $\tilde{h}$ and Q is help-free, op_2 has to be decided before any operation of p_3 at h, contradicting Lemma 3. Thus, both primitives have to be compare&swap.

By the same argument both compare&swap steps by p_1 and p_2 have the expected value that is equal to the current value in the designated memory location, and the new value is different from the expected. If it does not hold, either the local states of p_1 and the shared memory states after $h \circ p_1$ and $h \circ p_2 \circ p_1$ are identical or the local state of p_2 and the shared memory states after $h \circ p_2$ and $h \circ p_1 \circ p_2$ are identical.

Observation 4 *The primitive step of p_2 in Line 14 is a successful compare&swap, and the primitive step of p_1 in Line 15 is a failed compare&swap.*

Observation 5 *Immediately after Line 14 op_2 is decided before op_1.*

Lemma 7. *Immediately after Line 15 the order between op_1 and any operation of p_3 is not decided.*

Proof. By Lemma 3, the order between op_1 and any operation of p_3 is not decided before Line 14. Because Q is help-free the steps by p_2 cannot fix the order between op_1 and any operation of p_3. Thus, the only step that can fix the order of op_1 and some operation of p_3 is a step by p_1 at Line 15, i.e., a failed compare&swap.

Suppose that op_1 is decided before some operation op_3' of p_3 after Line 15. Let h be the history right before Line 14. Consider two histories $h' = h \circ p_2 \circ p_1$ and $h'' = h \circ p_2$. Let p_3 to solo run `pop` operations after h' until it completes operation op_3' and then, further, until `pop` operation returns $\perp$, i.e., the stack is empty. Since op_1 is decided before op_3', some completed operation op_3'' of p_3 has to return 1: if we now complete op_1 it should be linearized before op_3'. Now, let p_3 to perform after h'' the same number of operations as it did after h'. Since the local states of p_3 and the shared memory states after h' and h'' are identical (p_1 makes the failed compare&swap), op_3'' after h'' has to return 1 as after h'. Thus, op_1 is decided before op_3'' in h''. Since Q is help-free and p_1 does not take steps after h in h'', op_1 has to be decided before op_3'' before Line 14, contradicting Lemma 3.

Lemma 8. *At the end of the outer loop (Line 18) the order between op_1 and next $op_2 = \texttt{push}(id_2 + 1)$ is not yet decided.*

Proof. The operation op_2 is not started, thus, it cannot be decided before op_1 by Observation 1 (2).

Suppose that op_1 is decided before op_2. By Lemma 2 op_1 has to be decided before all operations of p_3, contradicting Lemma 7.

Thus after this iteration of the loop the two invariants hold (Observation 5 and Lemmas 7 and 8), and p_1 took at least one primitive step.

This way we build a history in which p_1 takes infinitely many steps, but op_1 is never completed. This contradicts the assumption that Q is wait-free.

Theorem 2. *In a system with at least three processes and primitives read, write and compare&swap there does not exist a wait-free and help-free stack implementation.*

4.2 Adding Fetch&Add

Now suppose that the implementation is allowed to additionally use fetch&add primitives. We prove that there is no wait-free and help-free stack implementation in a system with at least *four* processes.

Again, by contradiction, suppose that such an implementation Q exists. We build an infinite history h in which either p_1 or p_2 executes infinitely many failed compare&swap steps, yet it never completes its operation, contradicting wait-freedom. In h, processes p_1, p_2, p_3 and p_4 follow the following programs: for $1 \le i \le 2$, p_i tries to perform $op_i = \texttt{push}(i)$; p_3 applies an infinite sequence of operations $\texttt{push}(3), \texttt{push}(4), \texttt{push}(5), \ldots$; and p_4 is about to perform an infinite sequence of $\texttt{pop}()$ operations. The algorithm for constructing this history is given in Listing 1.2.

```
h ← ϵ
for i in 1..2:
   op_i ← push(i)
id_3 ← 3
```

```
5  while true:              // outer loop
6    op3 ← push(id3)
7    while true:            // inner loop
8      moved ← False
9      for i in 1..3:
10       if op_i is not decided before any op_j in h ∘ p_i:
11         h ← h ∘ p_i
12         moved ← True
13     if not moved:
14       break
15
16   h ← h ∘ p3
17   // let p_k be the process whose next primitive is compare&swap
18   h ← h ∘ p_k
19   while op3 is not completed:
20     h ← h ∘ p3
21   id3 ← id3 + 1
```

Listing 1.2: Constructing the history for the proof of Theorem 3

Similar to the proof of Theorem 2, we show that the following two invariants hold at the beginning of each iteration of the outer loop (Line 6):

- the order between any two operations among op_1, op_2 and op_3 is not decided;
- op_1 and op_2 are not decided before any operation of p_4;
- all the operations of p_3 prior to op_3 are decided before op_1 and op_2.

At the beginning of the first iteration, the invariants hold trivially, since none of op_i is started.

Observation 6 *The order between op_i and op_j for $1 \leq i \neq j \leq 3$ cannot be decided during (and right after) the inner loop (Lines 7-14).*

Proof. From the first invariant, op_i cannot be decided before op_j prior to the inner loop (Lines 7-14). Since Q is help-free, during the inner loop op_i can become decided before op_j only after a step by p_i which is impossible due to the check in Line 10.

Observation 7 *Process p_4 never takes a step in h.*

Lemma 9. *During (and right after) an execution of the inner loop (Lines 7-14) op_1, op_2 and op_3 cannot be decided before any operation of p_4.*

Proof. Suppose that during an execution of the inner loop op_1, op_2 or op_3 is decided before some operation of p_4.

At the beginning of the loop, none of op_1, op_2 and op_3 is decided before any operation of p_4: op_1 and op_2 are not decided because of the second invariant, while op_3 is not yet started. Suppose that during the execution of the inner loop some op_i becomes decided before some operations of p_4.

Let us look at the execution and find the first point in time when some op_k by p_k is decided before some operation op_4 of p_4. Using the same argument as in the proof of Lemma 3, we can show that op_k has to be decided before any other op_j contradicting Observation 6: we let p_4 run until the operation op_4 is completed and, further, while stack is not empty; op_4 becomes decided before op_j; by Transitivity Lemma 1, op_k is decided before op_j.

The proofs of the following two lemmas are identical to those of Lemmas 4 and 5.

Lemma 10. *For each i, $1 \leq i \leq 3$, op_i cannot be completed after the inner loop (Lines 7-14).*

Lemma 11. *The execution of the inner loop (Lines 7-14) is finite.*

Lemma 12. *For all i, j, $1 \leq i \neq j \leq 3$, op_i is decided before op_j in $h \circ p_i$.*

Proof. Consider an operation of process i. At the end of the inner loop op_i should be decided before some op_k in $h \circ p_i$, otherwise, p_i can make at least one more step during the inner loop. Thus, by Lemma 2 op_i should be decided before op_4, the first operation of p_4. Let p_4 run `pop` operations until one of them returns $\perp$, i.e., the stack is empty. Let this history be h'.

By Lemma 9, op_j is not decided before any operation of p_4 in h. Since Q is help-free and only p_i and p_4 takes steps in h' after h, op_j cannot be decided before any operation of p_4 in h', and, consequently, operations of p_4 cannot pop an argument of op_j. Since the operations of p_4 empty the stack, op_j must be linearized after them. Thus, op_4 is decided before op_j in h'. By Transitivity Lemma 1, op_i is decided before op_j in h'. Finally, since Q is help-free and only p_4 takes steps in h' after $h \circ p_i$, op_i is decided before op_j in $h \circ p_i$.

Lemma 13. *Immediately before Line 16 the following holds:*

1. *The next primitive step by p_i for $1 \leq i \leq 3$ is to the same memory location.*
2. *The next primitive step by p_i for $1 \leq i \leq 3$ is fetch&add with a non-zero argument or compare&swap for which the expected value is the value that appears in the designated location and the new value is different from the expected one.*

Proof. Suppose, that for some pair p_i and p_j the next steps are to different memory locations. We consider two histories $h' = h \circ p_i \circ p_j \circ \text{complete } op_i \circ \text{complete } op_j$ and $h'' = h \circ p_j \circ p_i \circ \text{complete } op_i \circ \text{complete } op_j$. By Lemma 12, after h', the two subsequent `pop` operations by p_4 should return first the argument of op_j and then the argument of op_i, while after h'' they should return the two values in the opposite order. This is impossible, since the local states of p_4 and the shared memory states after h' and h'' are identical. The same argument will apply if the next steps of some pair of processes are read primitives.

Suppose that the next primitive step of some p_i is a write. We take any other process p_j and build two histories: $h' = h \circ p_j \circ p_i \circ \text{complete } op_i$ and $h'' =$

$h \circ p_i \circ$ complete op_i. As in the proof of Lemma 6, p_i performs two pop operations (op'_i and op''_i) and p_j completes its operation after h': by Lemmas 1 and 12 op'_i and op''_i have to return the argument of op_i and the argument of op_j, respectively. The local states of p_i and the shared memory states after h' and h'' are identical, thus, op'_i and op''_i after h'' should return op_i and op_j. Hence, op_j has to be decided before op''_i in $\tilde{h} = h'' \circ$ perform $op'_i \circ$ perform op''_i. By Lemma 2, op_j is decided before any operation of p_4 in $\tilde{h}$. And, finally, since Q is help-free and p_j does not take steps in $\tilde{h}$ after h, op_j has to be decided before any operation of p_4 in h, contradicting Lemma 9.

A similar argument applies to the case when the next primitive step of some p_i is fetch&add with argument zero or compare&swap which expected value differs from the value in the designated location or the new value is equal to the expected. We take any other process p_j $(1 \leq j \leq 3)$ and build two histories $h' = h \circ p_i \circ p_j \circ$ complete p_j and $h'' = h \circ p_j \circ$ complete p_j. The proof for the previous case applies except that now the roles of p_i and p_j are swapped.

Lemma 14. *At most one out of p_1 and p_2 can have fetch&add as their next primitive step.*

Proof. Suppose that p_1 and p_2 have fetch&add as their next primitive step. Consider two histories $h' = h \circ p_1 \circ p_2$ and $h'' = h \circ p_2 \circ p_1$. From Lemma 12 op_1 is decided before op_2 in h', thus, by Lemma 2 op_1 is decided before the first operation op_4 of p_4. After h' p_4 performs k' pop operations until one of them returns $\perp$, i.e., the stack is empty. One pop has to return 1, because if we now complete op_1 it has to be linearized before op_4. The same with h'': p_4 performs k'' pops until one of them returns $\perp$. Since, the local states of p_4 and the shared memory states after h' and h'' are the same: two pop operations $pop_1()$ and $pop_2()$ of $k'(= k'')$ operations of p_4 after h' and h'' return 1 and 2.

Now, we show that op_1 and op_2 are decided before op_3 in h'. The same can be shown for h''. Consider a history $\tilde{h}$: h' continued with k' pop operations by p_4. By Lemma 12 op_1 is decided before op_3 in h'. From Lemma 9 and two facts that Q is help-free and op_3 does not make any steps after h in $\tilde{h}$, it follows that op_3 cannot be decided before any operation of p_4 in $\tilde{h}$ and, consequently, the operations of p_4 cannot pop an argument of op_3. Since k pops of op_4 empty the stack, op_3 has to linearize after them, making operation $pop_2()$ to be decided before op_3. Since $pop_2()$ returns 2 it has to be decided after op_2. By Transitivity Lemma 1, op_2 is decided before op_3 in $\tilde{h}$. Q is help-free and only p_4 takes steps after h', thus, op_2 is decided before op_3 in h'.

Now consider two histories $h' \circ$ complete op_3 and $h'' \circ$ complete op_3. In both of these histories, op_1 and op_2 are decided before op_3. After the first history let p_4 perform three pop operations and p_1 and p_2 complete push(1) and push(2): the three pops return id_3, 2 and 1, respectively. Analougously, after the second history three pop return id_3, 1 and 2. This is impossible, since the local states of p_4 and the memory states after these two histories are identical.

Observation 8 *From the previous lemma we know that the next primitive step of at least one process p_1 or p_2 is compare&swap. Let it be process p_k. By al-*

gorithm, p_3 takes a step at Line 16 changing the memory location either by fetch&add or by a successful compare&swap, thus, the next step of p_k at Line 18 should be a failed compare&swap.

Observation 9 *Immediately after Line 16, op_3 is decided before op_1 and op_2.*

Lemma 15. *Immediately after Line 18, op_1 and op_2 are not decided before any operation of p_4.*

Proof. We prove the claim for op_1, the case of op_2 is similar.

If p_2 took a step at Line 18, then by Lemma 9 and the fact that the steps by p_2 or p_3 cannot fix the order between op_1 and any operation of p_4 due to help-freedom, op_1 is not decided before any operation of p_4.

If p_1 took a step at Line 18, then by Lemma 9 and the fact that the steps by p_3 cannot fix the order between op_1 and any operation of p_4 due to help-freedom, the only step that could fix the order is a step by p_1 at Line 18, i.e., a failed compare&swap. Suppose that op_1 is decided before some op'_4 of p_4 after Line 18. We consider two histories $h' = h \circ p_3 \circ p_1$ and $h'' = h \circ p_3$. Let p_4 run solo after h' until it completes op'_4, and then further until some of its `pop` operations returns $\perp$, i.e., the stack becomes empty. Since op_1 is decided before op'_4, some completed operation op''_4 of p_4 has to return 1: if we now complete op_1 it has to be linearized before op'_4. Now, let p_4 to run the same number of `pop` operations after h''. Since the local states of p_4 and the shared memory states after h' and h'' are identical, op''_4 returns 1. Thus, op_1 is decided before op''_4 in h''. As Q is help-free and p_1 does not take steps after h in h'', op_1 has to be decided before op''_4 in h, contradicting Lemma 9.

Lemma 16. *At the end of the outer loop (Line 21), the order between any two operations among op_1, op_2 and the next $op_3 =$ **`push`**$(id_3 + 1)$ is not yet decided.*

Proof. The operation op_3 is not yet started, thus, it cannot be decided before op_i, $i = 1, 2$, by Observation 1 (2).

Suppose that op_i, $i = 1, 2$, is decided before op_j, then by Lemma 2 op_i has to be decided before all operations of p_4, contradicting Lemma 15.

We started with three invariants that hold before any iteration of the loop. By Observation 9 and Lemmas 15 and 16) the invariants hold after the iteration, and at least one of p_1 and p_2 made at least one primitive step.

This way we build a history in which one of op_1 and op_2 never completes its operation, even though it takes infinitely many steps. This contradicts the assumption that Q is wait-free.

Theorem 3. *In a system with at least four processes and primitives read, write, compare&swap and fetch&add, there does not exist a wait-free and help-free stack implementation.*

5 Related work

Helping is often observed in wait-free (e.g., [6,14,7,13]) and lock-free implementations (e.g., [3,12,9,11]): operations of a slow or crashed process may be finished by other processes. Typically, to benefit from helping, an operation should register a *descriptor* (either in a dedicated "announce" array or attached in the data items) that can be used by concurrent processes to help completing it.

We are aware of three alternative definitions of helping: (1) *linearization-based* by Censor-Hillel et al. [5] considered in this paper, (2) *valency-based* by Attiya et al. [4] and (3) *universal* by Attiya et al. [4].

Valency-based helping [4] captures helping through the values returned by the operations, which makes it quite restrictive. In particular, for stack, the definition cannot capture helping relations between two `push` operations. They distinguish *trivial* and *non-trivial* helping: for non-trivial helping, the operation that is being helped should return a data-structure-specific *non-trivial* (e.g., non-empty for stacks and queues) value. It is shown in [4] that any wait-free implementation of queue has non-trivial helping, while there exists a wait-free implementation of stack without non-trivial helping. This is an interesting result, given notorious attempts of showing that queue is in Common2 [2], i.e., that they can be implemented using reads, writes and 2-consensus objects, while stack has been shown to be in Common2 [1].

Attiya et al. [4] also introduce a very strong notion of helping — *universal helping* — which essentially boils down to requiring that every invoked operation eventually takes effect. This property is typically satisfied in universal constructions parameterized with object types. But most algorithms that involve helping in a more conventional (weaker) sense do not meet it, which makes the use of universal helping very limited.

Linearization-based helping [5] considered in this paper is based on the order between two operations in a possible linearization. Compared to valency-based definitions, this notion of helping operates on the linearization order and, thus, can be applied to all operations, not only to those that return (non-trivial) values. By relating "helping" to fixing positions in the linearization, this definiton appears to be more intuitive: one process helps another make a "progress", i.e., linearize earlier. Censor-Hillel et al. [5] also introduce two classes of data types: exact order types (queue as an example) and global view types (snapshot and counter as examples). They showed that no wait-free implementation of data types from these two classes can be help-free. By assuming stack to be exact order, they deduced that this kind of helping is required for wait-free stack implementations. In this paper, we show that stack is in fact not an exact order type, and give a direct proof of their claim.

6 Concluding remarks

In this paper, we give a direct proof that any wait-free implementation of stack in a system with read, write, comare&swap and fetch&add primitives is subject to linearization-based helping. This corrects a mistake in the indirect proof via exact order types in [5].

Let us come back to the original intuition of *helping* as a process performing work on behalf of other processes. One may say that linearization-based helping introduced by Censor-Hillel et al. and used in our paper does not adequately capture this intuition. For example, by examining the wait-free stack implementation by Afek et al. [1], we find out that none of the processes *explicitly* performs work for the others: to perform `pop`() a process goes down the stack from the current top until it reaches some value or the bottom of the stack; while to perform `push`(x) a process simply increments the top of the stack and deposits x there. But we just showed that any wait-free stack implementation has linearization-based helping, and indeed this algorithm has it. So we might think that valency-based helping is superior to linearization-based one, since the algorithm by Afek et al. does not have *non-trivial* valency-based helping. Nevertheless, the aforementioned algorithm has *trivial* valency-based helping, and, thus, the (quite unnatural) distinction between trivial and non-trivial helping seems to be chosen specifically to allow the algorithm by Afek et al. to be help-free.

A very interesting challenge is therefore to find a definition of linearization-based helping that would naturally reflect help-freedom of the algorithm by Afek et al., while queue does not have a wait-free and help-free implementation.

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 *PODC*, pages 159–170, 1993.
3. Maya Arbel-Raviv and Trevor Brown. Reuse, dont recycle: Transforming lock-free algorithms that throw away descriptors. In *DISC*, volume 91, pages 4:1–4:16, 2017.
4. Hagit Attiya, Armando Castañeda, and Danny Hendler. Nontrivial and universal helping for wait-free queues and stacks. In *OPODIS*, volume 46, 2016.
5. Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *PODC*, pages 241–250. ACM, 2015.
6. Panagiota Fatourou and Nikolaos D Kallimanis. A highly-efficient wait-free universal construction. In *SPAA*, pages 325–334. ACM, 2011.
7. Steven Feldman, Pierre Laborde, and Damian Dechev. A wait-free multi-word compare-and-swap operation. *IJPP*, 43(4):572–596, 2015.
8. Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):123–149, 1991.
9. Shane V Howley and Jeremy Jones. A non-blocking internal binary search tree. In *SPAA*, pages 161–171. ACM, 2012.
10. Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
11. Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82. ACM, 2002.
12. Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Notices*, volume 49, pages 317–328. ACM, 2014.
13. Yaqiong Peng and Zhiyu Hao. Fa-stack: A fast array-based stack with wait-free progress guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 2017.
14. Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *OPODIS*, pages 330–344. Springer, 2012.