Help!

Keren Censor-Hillel, Erez Petrank, Shahar Timnat
Department of Computer Science, Technion
{ckeren,erez,stimnat}@cs.technion.ac.il

August 2, 2015

Abstract

A fundamental challenge in designing concurrent data structures is obtaining efficient wait-free implementations, in which each operation completes regardless of the behavior of other operations in the system. The most common paradigm for guaranteeing wait-freedom is to employ a helping mechanism, in which, intuitively, fast processes help slow processes complete their operations. Curiously, despite its abundant use, to date, helping has not been formally defined nor was its necessity rigorously studied.

In this paper we initiate a rigorous study of the interaction between wait-freedom and helping. We start with presenting a formal definition of help, capturing the intuition of one thread helping another to make progress. Next, we present families of object types for which help is necessary in order to obtain wait-freedom. In other words, we prove that for some types there are no linearizable wait-free help-free implementations. In contrast, we show that other, simple types, can be implemented in a linearizable wait-free manner without employing help. Finally, we provide a universal strong primitive for implementing wait-free data structures without using help. Specifically, given a wait-free help-free fetch&cons object, one can implement any type in a wait-free help-free manner.

Keywords: Parallel Algorithms, Concurrent Data Structures, Progress Guarantees, Wait-Freedom, Helping.

1 Introduction

The era of multi-core architectures has been having a huge impact on software development: exploiting parallelism has become the main challenge of today's programming. With multiple processors communicating by accessing shared memory, the behavior of concurrent algorithms is measured by both *safety/correctness* and *progress* conditions.

Most of the code written today is lock-based, but this is shifting towards codes without locks [14]. The holy grail of designing concurrent data structures is in obtaining efficient *wait-free* implementations, with research dating back to some of the most important studies in distributed computing [10, 17, 21]. Wait-freedom refers to implementations in which each operation terminates in a finite, preferably small, number of steps and, in particular, without dependence on the behavior of other processes in the system.

While the goal is fundamental [4, 14], wait-free implementations are often more complicated than non wait-free designs such as *lock-free* implementations, which only require that some operation makes progress but not necessarily all. In practice, sometimes all processes complete their operations in a timely manner despite following a code that only guarantees lock-freedom. A line of work orthogonal to the one in this paper attempts to explain this behavior by the fact that the worst-case adversarial schedules are of low probability in practice. Such studies describe benevolent adversaries under which it is sufficient to design lock-free algorithms [2, 9, 15].

Nevertheless, wait-freedom captures progress against the worst possible behavior, and as such is vital for real-time systems. Previous work identifies relations of other properties of implementations to the possibility of being wait-free. For example, no universal construction can be wait-free and satisfy disjoint-access parallelism [6].

One common approach used in order to guarantee wait-freedom is to employ some *helping* mechanism [8, 12–14, 17, 19, 20, 23, 25, 26]. Loosely speaking, in helping mechanisms, apart from completing their own operation, processes perform some additional work whose goal is to facilitate the work of others. Curiously, despite being a crucial ingredient, whether explicitly or implicitly, in many implementations of concurrent data structures, the notion of helping has been lacking thorough study as a concept.

Intrigued by the tools needed in order to achieve wait-freedom, we offer in this work a rigorous study of the interaction between the helping paradigm and wait-freedom. In particular, we are interested in the following question: Does wait-freedom require help? To this end, we start by proposing a formal definition of help. The proposed definition is based on linearization order of histories of an implementation rather than on a semantic description. We give evidence that the proposed definition matches the intuitive notion. We then present and analyze properties of types for which any wait-free implementation necessitates help. Such types includes popular data structures such as the stack and the queue. In contrast, we show that other types can be implemented in a wait-free help-free manner. A natural example is an implementation of a set (with the INSERT, DELETE, and CONTAINS operations) with a bounded range of possible values.

1.1 Our Contributions

Our first contribution is definitial: we propose a formal definition of the existing intuitive concept of helping in Definition 3.3. Roughly speaking, a process p helps an operation of another process q if a step of p determines that q's operation is linearized before some other operation.

We note that there is some ambiguity in the literature regarding the concept of help; it is used informally to describe two different things. One usage of help is in the common case where processes of lock-free algorithms coordinate access to a shared location. Here, one process p_1 completes the (already ongoing) operation of another process p_2 in order to enable access to shared data and to allow p_1 to complete its operation. Barnes [5] uses this practice as a general technique to achieve lock-freedom. This is also the case for the queue of [22], where processes sometimes need to fix the tail pointer to point to the last node (and not the one before last) before they can execute their own operation. Loosely speaking, the purpose of the above practice is not "altruistic". A process fixes the tail pointer because otherwise it would not be able to execute its own operation.

This is very different from the usage of help in, e.g., UPDATE operations in [1], which perform embedded scans for the sole "altruistic" purpose of enabling concurrent SCAN operations. It also differs from reading a designated announcements array, whose sole purpose is to allow processes to ask other processes for help, such as in [17]. In [17], a process could have easily completed its operation without helping any other operation (by proposing to the consensus object used in this build a value that consists only the process's own value, without values of other processes viewed in the announcements array). Our definition of help deliberately excludes the former concept (where a process simply enables data access for its own operation), and captures only the latter "altruistic" form of help.

Having a formal notion of helping, we turn to study the interaction between wait-freedom and help. We look into characterizing properties of types that require help in any wait-free implementation. We define and analyze two general types of objects. The first type, which we call *Exact Order Types*, consists of types in which the order of operations affects the result of future operations. That is, for some operations sequences, every change in the order of operations influences the final state of the object. Natural examples of exact order types are FIFO queues and LIFO stacks.

We note that exact order types bear some similarity to previously defined objects, such as *perturbable objects* [18] and *class G objects* [7], since all definitions deal with an operation that needs to return different results in several different executions. However, these definitions are not equivalent. For example, queues are exact order types, but are not perturbable objects, while a max-register is perturbable but not exact order. We mention perturbable objects in Section 9.

The second type, which we call *Global View Types*, consists of types which support an operation that obtains the entire state of the object. Examples of global view types are snapshot objects, increment objects, and fetch&add. For instance, in an increment object that supports the operations GET and INCREMENT, the result of a GET depends on the exact number of preceding INCREMENTS. However, unlike the queue and stack, the result of an operation is not necessarily influenced by the internal order of previous operations. Notice that global view types are not equivalent to *readable objects* as defined by Ruppert [24], since for some global view types any applicable operation must change the state of the object. For example, a fetch&increment object is a global view type, but is not a readable object.

We prove that every wait-free implementation of any exact order type and any global view type requires help. Furthermore, when the CAS primitive is not available, we show that a max register [3] requires help even in order to provide lock-freedom.

Theorems 4.18, 5.23, 6.6 (rephrased): A linearizable implementation of a wait-free exact order type or a wait-free global view type using READ, WRITE, and CAS, or a lock-free max register using READ and WRITE, cannot be help-free.

We prove the above by constructing infinite executions in which some operation never completes unless helping occurs. This is done by carefully combining the definition of help with the attributes of the type.

We then show positive results, i.e., that some types can be implemented in a wait-free help-free manner. This is trivial for a *vacuous* type whose only operation is a NO-OP, but when CASes are available this also holds for a max register and for a *set* type, which supports the operations INSERT, DELETE and CONTAINS[1].

The proof that these types have wait-free help-free implementations can be generalized to additional types, provided they have an implementation in which every operation is linearized in a specific step of the same operation. Intuitively, these are implementations in which the result of an operation "does not depend too strongly" on past operations.

Naturally, the characterization of types which require help depends on the primitives being used, and while our results are generally stated for READ, WRITE, and CAS, we discuss additional primitives as well. In particular, we show that exact order types cannot be both help-free and wait-free even if the FETCH&ADD primitive is available, but the same statement is not true for global view types. Finally, we show that a fetch&cons primitive is universal for wait-free help-free objects. This means that given a wait-free help-free fetch&cons object, one can implement any type in a wait-free help-free manner.

1.2 Additional Related Work

Helping mechanisms come in different forms. Many wait-free implementations use a designated `announcement` array, with a slot for each process. Each process uses its slot to describe the operation it is currently seeking to execute, and other processes read this announcement and help complete the operation. This is perhaps the most widely used helping mechanism, appearing in specific designs, as well as in universal constructions [17], and in general techniques, such as for converting any lock-free data structure to a wait-free data structure [26].

But other forms of help exist. Consider, for example, the form of help that is used for the double-collect snapshot algorithm of [1]. In this wait-free snapshot object, each UPDATE operation starts by performing an embedded SCAN and adding it to the updated location. A SCAN operation op_1 that checks the object twice and sees no change can safely return this view. If a change has been observed, then the UPDATE operation op_2 that caused it also writes the view of its embedded SCAN, allowing op_1 to adopt this view and return it, despite the object being, perhaps constantly, changed. Thus, intuitively, the UPDATES help the SCANS.

2 Model and Definitions

We consider a standard shared memory setting with a fixed set of processes P. In each computation step, a process executes a single atomic primitive on a shared memory register, possibly preceded with some local computation. The set of atomic primitives contains READ, WRITE primitives, and usually also CAS. Where specifically mentioned, it is extended with the FETCH&ADD primitive.

A CAS primitive is defined by a triplet, consisting of a target register, an expected-value, and a new-value. When a CAS step is executed, the value stored in the target register is compared to the expected-value. If they are equal, the value in the target register is replaced with the new-value, and the Boolean value true is returned. In such a case we say that the CAS is *successful*. Otherwise, the shared memory remains unchanged, and false is returned. We stress that a CAS is executed atomically.

A FETCH&ADD primitive is defined by a target register and an integer value. An execution of the FETCH&ADD primitive atomically returns the value previously stored in the target register and replaces it with the sum of the previous value and the FETCH&ADD's integer value.

A *type* (e.g., a FIFO queue) is defined by a state machine, and is accessed via *operations*. An operation receives zero or more input parameters, and returns one result, which may be null. The state machine of a

[1] A degenerated set, in which the INSERT and DELETE operations do not return a boolean value indicating whether they succeeded can also be implemented without CASes.

type is a function that maps a state and an operation (including input parameters) to a new state and a result of the operation.

An *object*, is an implementation of a type using atomic primitives. An implementation specifies the primitives and local computation to be executed for each operation. The local computation can influence the next chosen primitive step. When the last primitive step of an operation is finished, the operation's result is computed locally and the operation is completed.

In the current work, we consider only executions of objects. Thus, a *program* of a process consists of operations on an object that the process should execute. The program may include local computations, and results of previous operations may affect the chosen future operations and their input parameters. A program can be finite (consisting of a finite number of operations) or infinite. This may also depend on the results of operations.

A *history* is a log of an execution (or a part of an execution) of a program. It consists of a finite or infinite sequence of computation steps. Each computation step is coupled with the specific operation that is being executed by the process that executed the step. The first step of an operation is also coupled with the input parameters of the operation, and the last step of an operation is also associated with the operation's result. A single computation step is also considered a history (of length one).

A *schedule* is a finite or infinite sequence of process ids. Given a schedule, an object, and a program for each process in P, a unique matching history corresponds. For a given history, a unique schedule corresponds. Given two histories, h_1, h_2, we denote by $h_1 \circ h_2$ the history derived from the concatenation of history h_2 after h_1. Given a program $prog$ for each process in P, and a history h, for each $p \in P$ we denote by $h \circ p$ the history derived from scheduling process p to take another single step following its program immediately after h.

The *set of histories created by an object* O is the set that consists of every history h created by an execution of any fixed set of processes P and any corresponding programs on object O, in any schedule S.

A history defines a partial order on the operations it includes. An operation op_1 is before an operation op_2 (denoted: $op_1 \prec op_2$) if op_1 is completed before op_2 begins. A sequential history is a history in which this order is a total order. A linearization [16] L of a history h is a sequence of operations (including their input parameters and results) such that 1) L includes all the operations that are completed in h, and may include operations that are started but are not completed in h, 2) the operations in L have the same input parameters as the operations in h, and also the same output results for operations that are completed in h, 3) for every two operations op_1 and op_2, if $op_1 \prec op_2$ in h, and op_2 is included in L, then $op_1 \prec op_2$ in L, and 4) L is consistent with the type definition of the object creating history h. An object O is a linearizable implementation of type T if each history in the set of histories created by O has a linearization.

Lock-freedom and wait-freedom are forms of progress guarantees. In the context of our work, they apply to objects (which are, as mentioned above, specific implementations of types). An object O is lock-free if there is no history h in the set of histories created by O such that 1) h is infinite and 2) only a finite number of operations is completed in h. That is, an object is lock-free if at least one of the executing processes must make progress and complete its operation in a finite number of steps. Wait-freedom is a strictly stronger progress guarantee. An object O is wait-free if there is no history h in the set of histories created by O such that 1) h includes an infinite number of steps by some process p and 2) the same process p completes only a finite number of operations in h. That is, O is wait-free if every process that is scheduled to run infinite computation steps must eventually complete its operation, regardless of the scheduling.

3 What is Help?

The conceptual contribution of this work is in establishing that many types cannot be implemented in a linearizable wait-free manner without employing a helping mechanism. To establish such a conclusion, it is necessary to accurately define help. In this section we discuss help intuitively, define it formally, and consider examples showing that the formal definition expresses the intuitive concept of help. Additionally, we will establish two general facts about help-free wait-free implementations.

3.1 Intuitive Discussion

Many wait-free algorithms employ an array with a designated entry for each process. A process announces in this array what operation it wishes to execute, and other processes that see this announcement might help and execute this operation for it. Such mechanisms are used in most wait-free universal constructions, dating back to [17] and many other constructions since. These mechanisms are probably the most explicit way to offer help, but not the only one possible. Considering help in a more general form, we find it helpful[2] to think of the following scenario.

Consider a system of three processes, p_1, p_2, p_3, and an object that implements a FIFO queue. The program of p_1 is ENQUEUE(1), the program of p_2 is ENQUEUE(2), and the program of p_3 is DEQUEUE$()$. First consider a schedule in which p_3 starts running solo until completing its operation. The result of the dequeue, regardless of the implementation of the FIFO queue, is null. If before scheduling p_3, we schedule p_1 and let it complete its operation, and only then let p_3 run and complete its own operation, p_3 will return 1. If we schedule p_1 to start executing its operation, and stop it at some point (possibly before its completion) and then run p_3 solo until completing its operation, it may return either null or 1. Hence, if we consider the execution of p_1 running solo, there is (at least) one computation step S in it, such that if we stop p_1 immediately before S and run p_3 solo, then p_3 returns null, and if we stop p_1 immediately after S and run p_3 solo, p_3 returns 1.

Similarly, if we consider the execution of p_2 running solo, there is (at least) one computation step that “flips” the value returned by p_3 when running solo from null to 2. We now consider a schedule that interleaves p_1 and p_2 until one of them completes. In any such execution, there is (at least) one computation step that “flips” the result of p_3 from null to either 1 or 2. If a step taken by p_2 “flips” the result of p_3 and causes it to return 1 (which is the value enqueued by p_1) we say that p_2 helped p_1. Similarly, if a step taken by p_1 “flips” the result of p_3 and causes it to return 2, then p_1 helped p_2.

This is the intuition behind the help notion that is defined below. Some known lock-free queue algorithms do not employ help, such as the lock-free queue of Michael and Scott [22]. However, we prove in Section 4 that any *wait-free* queue algorithm must employ help.

3.2 Help Definition

We say that an operation *op* belongs to history h if h contains at least one computation step of *op*. Note that *op* is a specific instance of an operation on an object, which has exactly one invocation, and one result. We say that the *owner* of *op* is the process that executes *op*.

Definition 3.1 (Linearization Function.) *We say that f is a linearization function over a set of histories H, if for every $h \in H$, $f(h)$ is a linearization of h.*

[2] Pun intended.

Definition 3.2 (Decided Operations Order.) *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 the set of histories H, if there exists no $s \in H$ such that h is a prefix of s and $op_2 \prec op_1$ in $f(s)$.*

Definition 3.3 (Help-Free Implementation.) *A set of histories H is without help, or help-free, if there exists a linearization function f over H such that 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 computation step γ is a step in the execution of op_1 by the owner of op_1*[3]*.*

An object is a help-free implementation, if the set of histories created by it is help-free.

To better understand this definition, consider an execution of an object. When considering two concurrent operations, the linearization of these operations dictates which operation comes first. The definition considers the specific step, γ, in which it is decided which operation comes first. In a help-free implementation, γ is always taken by the process whose operation is decided to be the one that comes first.

Consider the wait-free universal construction of Herlihy [17]. One of the phases in this construction is a wait-free reduction from a *fetch-and-cons* list to consensus. A fetch-and-cons (or a fetch-and-cons list) is a type that supports a single operation, *fetch-and-cons*, which receives a single input parameter, and outputs an ordered list of the parameters of all the previous invocations of *fetch-and-cons*. That is, conceptually, the state of a fetch-and-cons type is a list. A *fetch-and-cons* operation returns the current list, and adds (hereafter, cons) its input operation to the head of the list.

The reduction from fetch-and-cons to consensus is as follows. A special announce array, with a slot for each process, is used to store the input parameter of each ongoing *fetch-and-cons* operation. Thus, when a process desires to execute a *fetch-and-cons* operation, it first writes its input value to its slot in the announce array.

Next, the process reads the entire announce array. Using this information, it calculates a *goal* that consists of all the operations recently announced in the array. The process will attempt to cons *all* of these operations into the fetch-and-cons list. It reads the current state of the fetch-and-cons list, and appends this list to the end of its own goal (removing duplications.) Afterwards, the process starts executing (at most) n instances of consensus (n is the number of processes). In each instance of consensus, a process proposes its own process id.

The goal of the process that wins the consensus represents the updated state of the fetch-and-cons list. Thus, if the process wins a consensus instance, it returns immediately (as its own operation has definitely been applied). If it loses a consensus, it updates its goal again to be its original goal (minus duplications that already appear it the updated state) followed by the new list, which is the goal of the last winner. After participating in n instances of consensus, the process can safely return, since at least one of the winners in these instances already sees the process's operation in the announces array, and includes it in its goal.

This is a classic example of help. Wait-freedom is obtained due to the fact that the effect of process p winning an instance is adding to the list all the items it saw in the announce array, not merely its own item. To see that this algorithm is not help-free according to Definition 3.3 consider a system of three processes. Each process first announces its wanted item in the ANNOUNCE array, and then reads all of the array. Assume p_1's place in the array is before p_2's, but that p_2 writes to the announce array first. p_3 then reads the announce array and sees p_2's item. Then p_1 writes to the announce array, and afterwards continue to read the entire array.

[3]For readers familiar with the concept of strong linearization [11], we note that a set of histories can be strongly linearizable yet not help-free, and can also be help-free yet not strongly linearizable.

At this point p_2 is stalled, while p_1 and p_3 start competing in consensus. If the winner is p_1, then p_1's item is added to the list before p_2's item (since p_1's place in the list is before p_2's). If the winner is p_3, then the item of p_2 is added to the list, but the item of p_1 not as yet. Thus, a step of p_3 can decide that the item of p_2 is before that of p_1, and thus the fetch-and-cons operation of p_2 comes before that of p_1. This contradicts help-freedom.

A system of two processes. In general, help is not required in a system with only two processes. The universal construction of [17] is help-free in such a system, and can be used to implement any type in a help-free wait-free manner. Accordingly, we concentrate on proving that help is required for systems with at least three processes.

3.3 General Observations

In this subsection we point out two facts regarding the decided operations order (Definition 3.2) that are useful to prove that some types cannot be both wait-free and help-free. The first fact is true for non help-free implementations as well, as it is derived directly from the linearizability criteria. It states that for completed operations, the decided order must comply with the partial order a history defines, and for future operations, the decided order cannot contradict partial orders that may apply later on.

Observation 3.4 *In any history h:*
(1) Once an operation is completed it must be decided before all operations that have not yet started.
(2) While an operation has not yet started it cannot be decided before any operation of a different process.
(3) In particular, the order between two operations of two different processes cannot be decided as long as neither of these operations have started.

The second fact is an application of the first observation for help-free implementations.

Claim 3.5 *In a help-free implementation in a system that includes at least three processes, for a given history h and a linearizarion function f, 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 future (not yet started) operation of any process.*

Proof: Immediately following h, allow p_2 to run solo long enough to complete the execution of op_2. By Observation 3.4, op_2 must now be decided before any future operation. Thus, by transitivity, op_1 must be decided before any future operation as well. In a help-free implementation, op_1 cannot be decided before a different operation as a result of a step of p_2. Thus, op_1 must be decided before future operations already at h.

4 Exact Order Types

In this section we prove that some types cannot be implemented in a linearizable, wait-free, and help-free manner. Simply put: for some types, wait-freedom requires help. We first prove this result for systems that support only READ, WRITE, and CAS primitives. We later extend the proof to hold for systems that support the FETCH&ADD primitive as well. This section focuses on exact order types. Roughly speaking, these are types in which switching the order between two operations changes the results of future operations. An intuitive example for such a type is the FIFO queue. The exact location in which an item is enqueued is important, and will change the results of future dequeues operations.

In what follows we formally define exact order types. This definition uses the concept of a sequence of operations. 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 in S. We denote by $(S + op?)$ a sequence that contains S and possibly also the operation op. That is, $(S + op?)$ is in fact a set of sequences that contains S, and also sequences that are similar to S, except that a single operation op is inserted in somewhere between (or before or after) the operations of S.

Definition 4.1 (Exact Order Types.) *An exact order type t is a 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 at least one operation in $R(m)$, the result it returns in any execution in $W(n+1) \circ (R(m) + op?)$ differs from the result it returns in any execution in $W(n) \circ op \circ (R(m) + W_{n+1}?)$.*

Examples of such types are a queue, a stack, and the fetch-and-cons used in [17]. To gain some intuition about the definition, consider the queue. Let op be an ENQUEUE(1) operation, W be an infinite sequence of ENQUEUE(2) operations, and R be an infinite sequence of DEQUEUE operations. The queue is an exact order type, because the $(n+1)$-st dequeue returns a different result in any execution that starts with $n+1$ ENQUEUE(2) operations compared to any execution that starts with n ENQUEUE(2) operations and then an ENQUEUE(1).

More formally, let n be an integer, and set m to be $n+1$. Executions in $W(n+1) \circ (R(m) + op?)$ start with $n+1$ ENQUEUE(2) operations, followed by $n+1$ DEQUEUE operations. (There is possibly an ENQUEUE(1) somewhere between the dequeues, but not before any of the ENQUEUE(2).) Executions in $W(n) \circ op \circ (R(m) + W_{n+1}?)$ start with n ENQUEUE(2) operations, then an ENQUEUE(1) operation, and then $n+1$ DEQUEUE operations. (Again, there is possibly an ENQUEUE(2) somewhere between the dequeues.) From the specification of the FIFO queue, the last DEQUEUE must return a different result in the first case (in which it must return 2) than in the second case (in which it must return 1).

We now turn to prove that any exact order type cannot be both help-free and wait-free. Let Q be a linearizable, help-free implementation of an exact order type. The reader may find it helpful to consider a FIFO queue as a concrete example throughout the proof. We will prove that Q is not wait-free. For convenience, we assume Q is lock-free, as otherwise, it is not wait-free and we are done. Let op_1, W, and R be the operation and sequences of operations, respectively, guaranteed in the definition of exact order types. Consider a system of three processes, p_1, p_2, and p_3. The program of process p_1 is the operation op_1. The program of process p_2 is the infinite sequence W. The program of process p_3 is the (finite or infinite) sequence R. The operation of p_1 is op_1, an operation of p_2 is denoted op_2, and the first operation of p_3 is denoted op_3.

We start by proving two claims that are true for any execution of Q in which p_1, p_2, and p_3 follow their respective programs. These claims are the only ones that directly involve the definition of exact order types. The rest of the proof considers a specific execution, and builds on these two claims.

Claim 4.2 *Let h be a history such that the first n operations are already decided to be the first n operations of p_2 (which are $W(n)$), and p_3 has not yet taken any step. (Denote the $(n+1)$-st operation of p_2 by op_2.)*
(1.) If in h op_1 is decided before op_3, then the order between op_1 and op_2 is already decided.
(2.) Similarly, if in h op_2 is decided before op_3, then the order between op_1 and op_2 is already decided.

Proof: For convenience, we prove (1). The proof of (2) is symmetrical. Assume that in h op_1 is decided before op_3, and let m be the integer corresponding to n by the definition of exact order types. Immediately after h, let p_3 run in a solo execution until it completes exactly m operations. Denote the history after this solo execution of p_3 by h', and consider the linearization of h'.

The first n operations in the linearization must be $W(n)$. The linearization must also include exactly m operations of p_3 (which are $R(m)$), and somewhere before them, it must also include op_1. The linearization may or may not include op_2. There are two cases. If the $(n+1)$-st operation in the linearization is op_1, then the linearization is in $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$, while if the $n+1$-st operation in the linearization is op_2, then the linearization must be exactly $W(n+1) \circ op_1 \circ R(m)$ which is in $W(n+1) \circ (R(m) + op_1?)$. We claim that whichever is the case, the order between op_1 and op_2 is already decided in h'.

To see this, consider any continuation $h' \circ x$ of h'. Consider the linearization of $h' \circ x$. This linearization must also start with $W(n)$, must also include $R(m)$, and somewhere before $R(m)$ it must include op_1. It may or may not include op_2 somewhere before R_m. All the rest of the operations in $h' \circ x$ must be linearized after R_m, because they where not yet started when R_m was completed. Thus, the linearization of $h' \circ x$ must begin with a prefix that belongs to either $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$ or $W(n+1) \circ (R(m) + op_1?)$. Mark this prefix as Prefix(Lin($h' \circ x$)).

So far we have obtained that for every x such that $h' \circ x$ is a continuation of h', Prefix(Lin($h' \circ x$)) belongs to either $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$ or $W(n+1) \circ (R(m) + op_1?)$. The next step is to show that for every x, Prefix(Lin($h' \circ x$)) belongs to the same one of these sets. To do this, we consider the results of $R(m)$ in h'.

In h', the operations $R(m)$ are already completed, and their results are set. By definition of exact order types, these results cannot be consistent with both $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$ and $W(n+1) \circ (R(m) + op_1?)$. Thus, if the linearization of h' is in $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$, then the results of $R(m)$ mean that for every x, Prefix(Lin($h' \circ x$)) cannot be in $W(n+1) \circ (R(m) + op_1?)$, and thus must belong to $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$. Similarly, if the linearization of h' is in $W(n+1) \circ (R(m) + op_1?)$, then for every x, Prefix(Lin($h' \circ x$)) must be in $W(n+1) \circ (R(m) + op_1?)$ as well.

Since the order between op_1 and op_2 is the same for each continuation of h', it follows by definition that the order between op_1 and op_2 is already decided in h'. Since Q is a help-free implementation, then the order between op_1 and op_2 cannot be decided during the solo execution of p_3 which is the delta between h and h'. Thus, the order between op_1 and op_2 is already decided in h.

Claim 4.3 *Let h, h', and h'' be three histories, such that in all of them the first n operations are already decided to be the first n operations of p_2 (which are $W(n)$), and p_3 has not yet taken any step. (Denote the $(n+1)$-st operation of p_2 by op_2.) Furthermore, in h the order between op_1 and op_2 is not yet decided, in h' op_1 is decided before op_2, and in h'' op_2 is decided before op_1.*
(1.) h' and h'' are distinguishable by p_3.
(2.) h and h' are distinguishable by at least one of p_2 and p_3.
(3.) h and h'' are distinguishable by at least one of p_1 and p_3.

Remark 4.4 *(3.) is not needed in the proof, but is stated for completeness.*

Proof: Let m be the integer corresponding to n by the definition of exact order types. We start by proving (1). Since in h' op_1 is decided before op_2, then in h' op_1 must also be decided before op_3 by Claim 3.5. Assume that immediately after h' p_3 is run in a solo execution until it completes exactly m operations. The linearization of this execution must start with $W(n)$, followed by op_1. This linearization must also include the first m operations of p_3 (which are $R(m)$, and it may or may not include op_2. Thus, the linearization must be in $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$.

Now assume that immediately after h'' p_3 is run in a solo execution until it completes exactly m operations. This time, the linearization must be in $W(n+1) \circ (R(m) + op_1?)$. By the definition of exact order types, there is at least one operation in $R(m)$, that is, at least one operation of p_3, which returns a different result in these two executions. Thus, h' and h'' are *distinguishable* by process p_3.

```
1:  h = ϵ;
2:  op1 = the single operation of p1;
3:  while (true)                                                  ▷ main loop
4:      op2 = the first uncompleted operation of p2;
5:      while (true)                                              ▷ inner loop
6:          if op1 is not decided before op2 in h ∘ p1
7:              h = h ∘ p1;
8:              continue;                                         ▷ goto line 5
9:          if op2 is not decided before op1 in h ∘ p2
10:             h = h ∘ p2;
11:             continue;                                         ▷ goto line 5
12:         break;                                                ▷ goto line 13
13:     h = h ∘ p2;                          ▷ this step will be proved to be a CAS
14:     h = h ∘ p1;                   ▷ this step will be proved to be a failed CAS
15:     while (op2 is not completed in h)          ▷ run p2 until op2 is completed
16:         h = h ∘ p2;
```

Figure 1: The algorithm for constructing the history in the proof of Theorem 4.18.

We turn to prove (2). Assume that immediately after h' p_2 is run until it completes op_2, and then p_3 is run in a solo execution until it completes exactly m operations. The linearization of this execution must be exactly $W(n) \circ op_1 \circ W_{n+1} \circ R(m)$ which is in $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$.

Now assume that immediately after h p_2 is run until it completes op_2 and then p_3 is run in a solo execution until it completes exactly m operations. At the point in time exactly after op_2 is completed, and exactly before p_3 starts executing op_3, op_2 is decided before op_3 (Observation 3.4). Thus, by Claim 4.2, the order between op_1 and op_2 is already decided. Since the order is not decided in h, the implementation is wait-free, and p_1 has not taken another step since h, it follows that op_2 must be decided before op_1.

In other words, in the execution in which after h p_2 completes op_2 and then p_3 completes exactly m operations, op_2, which is W_{n+1}, is decided before both op_3 and op_1. Thus, the linearization of this execution must be in $W(n+1) \circ (R(m) + op_1?)$.

By the definition of exact order types, there is at least one operation in $R(m)$, that is, at least one operation of p_3, which returns a different result in these two executions. Thus, h and h' are *distinguishable* by at least one of the processes p_2 and p_3. The proof of (3) is similar.

In the rest of the proof of the main theorem we build an infinite history h, such that the processes p_1, p_2, and p_3 follow their respective programs, and p_1 executes infinitely many (failed) CAS steps, yet never completes its operation, contradicting wait-freedom. The algorithm for constructing this history is depicted in Figure 1. In lines 5–12, p_1 and p_2 are scheduled to run their programs as long as it is not yet decided which of their operations comes first. Afterwards, the execution of Q is in a critical point. If p_1 were to take a step, then op_1 will be decided before op_2, and if p_2 were to take a step, then op_2 will be decided before op_1. We prove using indistinguishability arguments, that the next step by both p_1 and p_2 is a CAS (and that both steps access the same target register). Next (line 13), p_2 executes its CAS, and then (line 14) p_1 attempts a CAS as well, which is going to fail. Afterwards, p_2 is scheduled to complete its operation, and then the above is repeated with p_2's next operation.

It is shown that in iteration $n+1$ of the algorithm for constructing h, the n first operations are already decided to be the first n operations of p_2 (that is, $W(n)$), and iteration $n+1$ is a "competition" between op_1 and W_{n+1}. Exact order types are used to show that the two possible results of this competition must be distinguishable from one another.

We prove a series of claims on the execution of history h, which is a history of object Q. Most claims refer to the state of the execution of Q in specific points in the execution, described by a corresponding line in the algorithm given in Figure 1. These claims are proved by induction, where the induction variable is the iteration number of the main loop (lines 3–16). The induction hypothesis is that claims (4.5–4.16) are correct. Claim 4.5 is the only one to use the induction hypothesis directly. The other claims follow from Claim 4.5.

Claim 4.5 *Immediately after line 4, it holds that 1) the order between op_1 and op_2 is not yet decided, and 2) all the operations of p_2 prior to op_2 are decided before op_1.*

Proof: For the first iteration of the main loop, this is trivial because h is empty (Observation 3.4). For iteration $i \geq 2$, it follows from the induction hypothesis, Observation 4.13, and Claim 4.16.

Observation 4.6 *The order between op_1 and op_2 cannot be decided during the inner loop (lines 5–12).*

This follows from the fact that Q is help-free, and from inspecting the conditions in lines 6 and 9.

Observation 4.7 *Process p_3 never takes a step in h.*

Claim 4.8 *The order between op_1 and op_2 must be decided before any one of op_1 and op_2 is completed.*

Proof: If op_1 is completed, then op_1 must be decided before all future operations of p_3 (Observation 3.4). All the operations of p_2 prior to op_2 are already decided before op_1 (Claim 4.5), and by Observation 4.7, p_3 hasn't taken any steps. Thus, by Claim 4.2, the order between op_1 and op_2 is already decided.

Similarly, if op_2 is completed, then op_2 must be decided before all future operations of p_3 (Observation 3.4). Again, all the operations of p_2 prior to op_2 are already decided before op_1 (Claim 4.5), and by Observation 4.7, p_3 hasn't taken any steps. Thus, by Claim 4.2, the order between op_1 and op_2 is already decided.

Claim 4.9 *The execution of the inner loop (lines 5–12) is finite.*

Proof: By combining Observation 4.6 and Claim 4.8, no operation in Q is completed in h during the execution of the inner loop. Since Q is lock-free, and each loop iteration adds a single step to h, this cannot last infinitely.

Observation 4.10 *Immediately before line 13 op_1 is decided before op_2 in $h \circ p_1$, op_2 is decided before op_1 in $h \circ p_2$, and, hence, the order of op_1 and op_2 is not decided in h.*

From observing the code, the inner loop exits and line 13 is reached only if the next step of either p_1 or p_2 will decide the order. Since the queue algorithm is help-free, in $h \circ p_1$, op_1 is decided before op_2, and in $h \circ p_2$, op_2 is decided before op_1.

Claim 4.11 *Immediately before line 13 the following holds.*
(1.) The next primitive step in the programs of both p_1 and p_2 is to the same memory location.
(2.) The next primitive step in the programs of both p_1 and p_2 is a CAS.
(3.) The expected-value of both the CAS *operations of p_1 and p_2 is the value that appears in the designated address.*
(4.) The new-value of both the CAS *operations is different than the expected-value.*

Proof: By Observation 4.10, in $h \circ p_1$, op_1 is decided before op_2. It follows that op_1 is decided before op_2 in $h \circ p_1 \circ p_2$ as well. Similarly, op_2 is decided before op_1 in $h \circ p_2 \circ p_1$. By Claim 4.3 (1), it follows that $h \circ p_1 \circ p_2$ must be *distinguishable* from $h \circ p_2 \circ p_1$ by process p_3. It immediately follows that the next primitive step of both p_1 and p_2 is to the same memory location. Furthermore, the next step of both p_1 and p_2 cannot be a READ primitive. Also, it cannot be a CAS that does not change the shared memory, i.e., a CAS in which the expected-value is different than the value in the target address, or a CAS in which the expected-value and new-value are the same.

Thus, the next step by both p_1 and p_2 is either a WRITE primitive or a CAS which satisfies conditions (3) and (4) of the claim. It remains to show the next step is not a WRITE. Assume by way of contradiction the next step by p_1 is a WRITE. Then, $h \circ p_1$ is indistinguishable from $h \circ p_2 \circ p_1$ to all process excluding p_2, again contradicting Claim 4.3 (1). A similar argument also shows that the next step of p_2 cannot be a WRITE.

Claim 4.11 immediately implies:

Corollary 4.12 *The primitive step p_2 takes in line 13 is a successful* CAS*, and the primitive step p_1 takes in line 14 is a failed* CAS.

Observation 4.13 *Immediately after line 13, op_2 is decided before op_1.*

This follows immediately from Observation 4.10, and from line 13 of the algorithm for constructing h. Next, for convenience, we denote the first operation of p_3 as op_3.

Claim 4.14 *Immediately before line 13, the order between op_1 and op_3 is not yet decided.*

Proof: Process p_3 has not yet taken any steps (Observation 4.7), and thus its operation cannot be decided before op_1 (Observation 3.4). Assume by way of contradiction that op_1 is decided before op_3. All the operations of p_2 prior to op_2 are already decided before op_1 (Claim 4.5) and thus by Claim 4.2, the order between op_1 and op_2 is already decided. But the order between op_1 and op_2 is not yet decided before line 13 (Claim 4.5 and Observation 4.6), yielding contradiction.

Claim 4.15 *Immediately after completing line 16, the order between op_1 and op_3 is not yet decided.*

Proof: By Claim 4.14, the order between op_1 and op_3 is not yet decided before line 13. Steps by p_2 cannot decide the order between op_1 and op_3 in a help-free algorithm, and thus the only step which could potentially decide the order until after line 16 is the step p_1 takes in line 14. Assume by way of contradiction this step decides the order between op_1 and op_3.

If this step decides the order between op_1 and op_3 then after this step op_1 must be decided before op_3. By Corollary 4.12, this step is a failed CAS. Thus, the state immediately before this step and the state immediately after this step are indistinguishable to all processes other than p_1. This contradicts Claim 4.3 (2).

Claim 4.16 *Immediately after line 16, the order between op_1 and the operation of p_2 following op_2 is not yet decided.*

Proof: The operation of p_2 following op_2 has not yet begun, and thus it cannot be decided before op_1 (Observation 3.4). Assume by contradiction that op_1 is decided before the next operation of p_2. Thus, by Claim 3.5, op_1 must be decided before all future operations of p_3, including op_3. But by Claim 4.15, op_1 is not yet decided before op_3, yielding a contradiction.

Corollary 4.17 *Q is not wait-free.*

Proof: By Claim 4.9, each execution of the inner loop is finite. Thus, there are infinitely many executions of the main loop. In each such execution, p_1 takes at least a single step in line 14. Thus p_1 takes infinitely many steps. Yet, by combining Claims 4.5, and 4.8, op_1 is not completed in any iteration of the main loop, which implies it is never completed. Thus, Q is not wait-free.

Since the assumptions on Q were that it is linearizable, help-free, and lock-free, we can rephrase Corollary 4.17 as follows.

Theorem 4.18 *A wait-free linearizable implementation of an exact order type cannot be help-free.*

It is interesting to note that in history h built in this proof, process p_3 never takes a step. Nevertheless, its existence is necessary for the proof. History h demonstrates that in a lock-free help-free linearizable implementation of an exact order type, a process may fail a CAS infinitely many times, while competing processes complete infinitely many operations. This is indeed a possible scenario in the lock-free help-free linearizable queue of Michael and Scott [22], where a process may never successfully ENQUEUE due to infinitely many other ENQUEUE operations.

4.1 Generalizing the Proof To Cover the Fetch&Add Primitive

In the proof of Theorem 4.18, we assumed the allowed primitives were READ, WRITE, and CAS. Another primitive, not as widely supported in real machines, is the FETCH&ADD primitive. As we shall see in Section 5, when it comes to the question of wait-free help-free types, the FETCH&ADD primitive adds strength to the computation, in the sense that some types that cannot be implemented in a wait-free help-free manner using only the READ, WRITE, and CAS primitives, can be implemented in a wait-free help-free manner if the FETCH&ADD primitive is allowed (An example for such a type is the fetch&add type itself). However, in this subsection we claim that types such as the queue and stack cannot be implemented in a linearizable, help-free, wait-free manner, even if FETCH&ADD is available. In what follows we give this proof.

If we allow the FETCH&ADD primitive, yet leave the proof of Theorem 4.18 unchanged, the proof fails since Claim 4.11 fails. Originally, Claim 4.11 shows that immediately before line 13, the next steps in the programs of both p_1 and p_2 are CAS primitives to the same location. Furthermore, the claim shows that each of these CAS operations, if executed immediately, will modify the data structure. That is, the expected-value is the same as the value in the target address, and the new-value is different than the expected-value. Claim 4.11 proves this by elimination: it proves that the next steps of both p_1 and p_2 cannot be a READ, a WRITE, or a CAS that doesn't modify the data structure. This remains true when FETCH&ADD is allowed. However, a CAS that changes the data structure ceases to be the only remaining alternative.

We claim that immediately before line 13, it is impossible that the next steps of both p_1 and p_2 are FETCH&ADD, because then $h \circ p_1 \circ p_2$ is indistinguishable from $h \circ p_2 \circ p_1$ by p_3. After any of these two sequences, the order between op_1 and op_2 must be decided, and thus the first one of them must also be decided before all the future operations of p_3 (Claim 3.5). Thus, a long enough solo execution of p_3 will reveal which of one of op_1 and op_2 is linearized first, and the indistinguishability yields a contradiction.

Thus, immediately before line 13 it is impossible that the next steps of both p_1 and p_2 are FETCH&ADD. However, it is possible indeed that one of them is FETCH&ADD, and the other is a CAS. This foils the rest of the proof. To circumvent this problem, we add an extra process, denoted p_0. The program of p_0 consists of a single operation, denoted op_0.

A solo execution of p_3 should return different results if op_0 is executed, op_1 is executed, or op_2 is executed. For instance, in the case of the FIFO queue, op_0 can be ENQUEUE(0), op_1 ENQUEUE(1), and

op_2 ENQUEUE(2). As before, the program of p_2 is an infinite sequence of ENQUEUE(2) operations. The program of p_3 is an infinite sequence of DEQUEUE operations. In these settings, three process (p_0, p_1, p_2) "compete" to linearize their operation first in each iteration of the main-loop.

The inner loop (originally lines 5–12) is modified to advance the three processes. The conditions in lines 6 and 9 need not be changed; it is enough to check for each operation that it is not decided before one of the other two: at the first time an operation of op_0, op_1 and op_2 is decided before another one of these three operations, it is also decided before the last one. To see this, assume without loss of generality that at the first time such a decision is made, op_1 is decided before op_2. By Claim 3.5, it must also be decided before future operations of p_3. Run p_3 long enough, and see which operation comes first. Since op_0 is not yet decided before op_1, and cannot be decided to be before it during a solo execution of p_3, then p_3 must witness that op_1 is the first linearized operation, which implies that op_1 is decided before op_0 as well.

Thus, after the inner loop, the order between op_0, op_1, and op_2 is not yet decided, but if any of the processes p_0, p_1 or p_2 takes a step, its operation will be decided before the other two. As before, the next step of all of them must be to the same memory location. As before, their next steps cannot be a READ, a WRITE, or a CAS that does not change the memory. It is possible that the next step of one of them is FETCH&ADD, but as shown above, it is impossible that the next step of *two* of them is FETCH&ADD. Thus, the next step of at least one out of p_0 and p_1 must be a CAS. Next, we schedule p_2 to take a step, and afterwards we schedule p_0 or p_1 (we choose the one whose next step is a CAS) to take is step. This step must be a failed CAS.

The proof continues similarly as before. The failed CAS cannot decide an operation before op_3 because of indistinguishability. Process p_2 runs to complete op_2, and the above is repeated with the next operation of p_2. In each iteration of the main loop, at least one of p_0 and p_1 takes a single step, but neither op_0 or op_1 is ever completed, and thus the data structure is not wait-free. The conclusion is that a queue (or a stack) cannot be linearizable, help-free, and wait-free, even if the FETCH&ADD primitive as available.

To generalize this result to a family of types, we need to slightly strengthen the requirements of exact order types. The current definition of exact order types implicitly implies a repeated "competition" between two threads, the result of which can be witnessed by a third thread. Extending this definition to imply a repeated competition of three threads yields the following definition.

Definition 4.19 (Extended Exact Order Types.) *An extended exact order type T is a type for which there exist two operations op_0 and op_1, 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 at least one operation in $R(m)$, the value it returns in any execution in $W(n+1) \circ ((R(m) + op_0?) + op_1?)$ differs from the value it returns in any execution in $W(n) \circ op_0 \circ ((R(m) + W_{n+1}?) + op_1?)$, and both differ from the value it returns in any execution in $W(n) \circ op_1 \circ ((R(m) + W_{n+1}?) + op_0?)$.*

5 Global View Types

In this section we investigate a different set of types, that can also not be obtained in a wait-free manner without using help. These are types that support an operation that returns some kind of a global view. We start by addressing a specific example: a single-scanner snapshot. We later identify accurately what other types belong to this group. The technique of the proof used here is similar to that of Section 4, but the details are different and more complicated.

The single scanner snapshot type supports two operations: UPDATE and SCAN. Each process is associated with a single register entry, which is initially set to $\perp$. An UPDATE operation modifies the value of the register associated with the updater, and a SCAN operation returns an atomic view (snapshot) of all the registers. This variant is referred to as a single-writer snapshot, unlike a mulit-writer snapshot object that allows any process to write to any of the shared registers. In a single scanner snapshot, only a single SCAN operation is allowed at any given moment[4].

Let S be a linearizable, help-free implementation of a single scanner snapshot. We prove that S is not wait-free. For convenience, we assume S is lock-free, as otherwise, it is not wait-free and we are done. Consider a system of three processes, p_1, p_2, and p_3. The program of p_1 is a single UPDATE(0) operation, the program of p_2 is an infinite sequence alternating between UPDATE(0) and UPDATE(1) operations, and the program of process p_3 is an infinite sequence of SCAN operations.

Again, we build an infinite history h, such that processes p_1, p_2, and p_3 follow their respective programs. This time, we show that in h either p_1 executes infinitely many (failed) CAS steps, yet never completes its operation (as before), or alternatively, that starting at some point, neither p_1 nor p_2 complete any more operations, but at least one of them executes infinitely many steps.

The algorithm for constructing this history is depicted in Figure 2. In every iteration, the operations of p_1, p_2, p_3 are denoted op_1, op_2, op_3 respectively. In lines 6–13, processes p_1 and p_2 are scheduled to run their programs as long as neither op_1 nor op_2 is decided before op_3. After the loop is ended, if p_1 takes another step op_1 will be decided before op_3, and if p_2 takes another step then op_2 will be decided before op_3.

Then, in lines 14–15, p_3 is run as much as possible without changing the property achieved at the end of the previous loop. That is, when the loop of lines 14–15 is stopped, it is still true that 1) if p_1 takes another step then op_1 will be decided before op_3, and 2) if p_2 takes another step then op_2 will be decided before op_3. However, if p_3 will take another step, then at least one of (1) and (2) will no longer hold.

Now, the execution is divided into two cases. The first possibility is that if p_3 takes another step, both (1) and (2) will cease to hold simultaneously. In this case, similarly to the proof of Theorem 4.18, we show that both the CAS operations of p_1 and p_2 are to the same address, we allow p_2 to successfully executes its CAS, and let p_1 attempt its CAS and fail. Afterwards both op_2 and op_3 are completed, and we repeat the process with the next operations of p_2 and p_3.

The other possibility is that the next step of p_3 only causes one of the conditions (1) and (2) to cease to hold. Then, we allow p_3 to take the next step, and afterwards schedule the process (either p_1 or p_2) that can take a step without causing its operation to be decided before op_3. We prove this step is not a "real" progress, and cannot be the last step in the operation. Afterwards we allow op_3 to be completed, and repeat the process with the next operation of p_3.

Throughout the proof we avoid using the fact that a SCAN (op_3) returns a different result when it is linearized after op_1 and before op_2 compared to when it is linearized before op_2 and after op_1. We rely only on the fact that op_3 returns three different results if it is linearized before both UPDATE operations, before one of them, or after both. This more general approach slightly complicates the proof in a few places, but it makes the proof hold for additional types. In particular, this way the proof also holds for an increment object.

We use a similar inductive process as we did when proving Theorem 4.18: we prove a series of claims on the execution of history h, which is a history of object S. These claims are proved by induction, where the induction variable is the iteration number of the main loop (lines 2–27). The induction hypothesis is that claims (5.1–5.16) are correct. Claim 5.1 is the only one to use the induction hypothesis directly, while the other claims follow from it.

[4]Formally, the type is a snapshot, and a single-scanner implementation is a constrained implementation of it, in the sense that its correctness is only guaranteed as long as no two SCAN operations are executed concurrently.

1: $h = \epsilon$;
2: while (true) ▷ main loop
3: op_1 = the first uncompleted operation of p_1;
4: op_2 = the first uncompleted operation of p_2;
5: op_3 = the first uncompleted operation of p_3; ▷ scan operation
6: while (true) ▷ first inner loop
7: if op_1 is not decided before op_3 in $h \circ p_1$
8: $h = h \circ p_1$;
9: continue; ▷ goto line 6
10: if op_2 is not decided before op_3 in $h \circ p_2$
11: $h = h \circ p_2$;
12: continue; ▷ goto line 6
13: break; ▷ goto line 14
14: while (op_1 is decided before op_3 in $h \circ p_3 \circ p_1$ and op_2 is decided before op_3 in $h \circ p_3 \circ p_2$) ▷ second inner loop
15: $h = h \circ p_3$
16: if (op_1 is not decided before op_3 in $h \circ p_3 \circ p_1$ and op_2 is not decided before op_3 in $h \circ p_3 \circ p_2$)
17: $h = h \circ p_2$; ▷ this step will be proved to be a CAS
18: $h = h \circ p_1$; ▷ this step will be proved to be a failed CAS
19: while (op_2 is not completed in h) ▷ run p_2 until op_2 is completed
20: $h = h \circ p_2$;
21: else
22: Let $k \in \{1, 2\}$ satisfy op_k is not decided before op_3 in $h \circ p_3 \circ p_k$
23: Let $j \in \{1, 2\}$ satisfy op_j is decided before op_3 in $h \circ p_3 \circ p_j$
24: $h = h \circ p_3$;
25: $h = h \circ p_k$;
26: while (op_3 is not completed in h) ▷ run p_3 until op_3 is completed
27: $h = h \circ p_3$;

Figure 2: The algorithm for constructing the history in the proof of Theorem 5.19.

Claim 5.1 *Immediately after line 5, it holds that 1) the operation op_3 has not yet started, 2) the order between op_1 and op_3 is not yet decided, 3) the order between op_2 and op_3 is not yet decided.*

Proof: For the first iteration, none of the operations has started, thus the claim holds by Observation 3.4. For iteration $i \geq 2$, the claim follows from the induction hypothesis and Claim 5.16.

Claim 5.2 *Immediately before line 14, it holds that 1) the operation op_1 is not decided before any operation by either p_2 or p_3, 2) the operation op_2 is not decided before any operation by either p_1 or p_3, and 3) the operation op_3 is not decided before any operation by either p_1 or p_2.*

Loosely speaking, this claim states that no new ordering is decided during the execution of the first inner loop (lines 6–13).

Proof: By Claim 5.1(1), op_3 has not yet started after line 5. Since p_3 never advances in lines 6–13 then op_3 has not yet started immediately before line 14. Thus, op_3 cannot be decided before any operation of a different process (Observation 3.4), and we obtain (3).

We now turn to prove (1). First, we observe that before line 14, op_1 is not decided before op_3: by Claim 5.1(2), the operation op_1 is not decided before op_3 after line 5; the condition in line 7 guarantees that op_1 is not decided before op_3 as result of line 8, and the fact that the algorithm is help-free guarantees op_1 is not decided before op_3 as result of line 11.

Second, we claim op_1 is not decided before any operation of p_2. Assume by way of contradiction that op_1 is decided before an operation op of p_2. Thus, op_1 must be decided before all future operations of p_3 (Claim 3.5), including op_3. We just proved that op_1 is not decided before op_3, yielding a contradiction. Therefore, op_1 cannot be decided before any operation of p_2.

Finally, we claim that op_1 is not decided before any future operation of p_3. Assume by way of contradiction that op_1 is decided before an operation op of p_3. Using again Claim 3.5, op_1 must be decided before all future operations of p_2, yielding contradiction.

Thus, we have shown that immediately before line 14, the operation op_1 is not decided before op_3, not decided before any operation of p_2, and not decided before any (future) operation of p_3 as well, and (1) is proved. Condition (2) is proven the same way as (1).

Claim 5.3 *No operation in h is completed during the execution of the first inner loop (lines 6–13).*

Proof: By Claim 5.2, neither op_1 nor op_2 are decided before op_3 immediately before line 14. However, at the same point, op_3 has not yet begun (by Claim 5.1 and observing the code). If op_1 (or op_2) were completed immediately before line 14, then by Observation 3.4, it must have been decided before the future operation op_3. Thus, neither op_1 nor op_2 are completed immediately before line 14. Since only p_1 and p_2 take steps during the first inner loop and they do not complete their operations, it follows that no operation is completed.

Claim 5.4 *No operation in h is completed during the execution of the second inner loop (lines 14–15).*

Proof: By Claim 5.2, neither op_1 nor op_2 are decided before op_3 immediately before line 14. The operation op_3 itself has not yet begun before line 14. The condition in line 14 guarantees that op_3 will not be decided before op_1 or op_2 during the execution of the second inner loop, since after the second inner loop is over, a single step by p_1 (p_2) will decide op_1 (op_2) before op_3. Thus, after the second inner loop, the order between op_3 and op_1 and the order between op_3 and op_2 are not yet decided. In what follows we show that op_3 cannot be completed before these orders are decided, and thus reach the conclusion that op_3 cannot complete during the execution of the second inner loop.

The result of op_3 depends on the orders between op_3 and op_2, and between op_3 and op_1: the operation op_3 returns a certain result if op_3 is linearized before both op_1 and op_2, a different result if op_3 is linearized after both the other operations, and yet a different result than both previous results if op_3 is linearized before only one of op_1 and op_2.

It follows that if the result of op_3 is consistent with none of op_1 and op_2 linearized before it, then op_3 must already be decided before both op_1 and op_2. If the result is consistent with both op_1 and op_2 being linearized after op_3, then both must already be decided before op_3. Next, we claim that if the result is consistent with op_3 being linearized before exactly one of op_1 and op_2, then it must already be decided before each one. Assume by way of contradiction op_3 is not yet decided before either op_1 or op_2, but returns a result consistent with being linearized before exactly one of them.

According to the condition of line 14, after the second loop is completed, at least one of op_1 and op_2 will be decided before op_3 if its owner process will take one step. Let the owner take this step, and its operation (either op_1 or op_2) is now decided before op_3. Thus, op_3 must now be decided before the other operation (one of op_1 and op_2). However, in a help-free implementation op_3 cannot be decided before another operation as a result of a step taken by a process other than p_3, and thus op_3 must have been decided before either op_1 or op_2 before the second inner loop was completed, which yields a contradiction.

To conclude, op_3 cannot be completed before the order is decided, which means op_3 cannot be completed during the second inner loop. No other operation can be completed during the second inner loop as p_3 is the only process that advances in that loop.

Claim 5.5 *The executions of the first inner loop (lines 6–13) and second inner loop (lines 14–15) are finite.*

Proof: In each iteration of the first and second inner loops, a process advances a step in h. The history h is a history of the lock-free object S, and thus an infinite execution without completing an operation is impossible. By Claims 5.3 and 5.4, no operation is completed in these two loops, and thus their execution must be finite.

Claim 5.6 *Immediately before line 16, it holds that 1) the operation op_1 is not decided before any operation by either p_2 or p_3, and 2) the operation op_2 is not decided before any operation by either p_1 or p_3.*

Proof: These conditions have already been shown to hold before line 14 (Claim 5.2). The only process that takes steps in the second inner loop (lines 14–15) is p_3. In a help-free algorithm, steps by p_3 can only decide an operation of p_3 before any other operation.

Observation 5.7 *If the condition in line 16 is true, then immediately before line 17, the operation op_1 is decided before op_3 in $h \circ p_1$, the operation op_1 is not decided before op_3 in $h \circ p_3 \circ p_1$, the operation op_2 is decided before op_3 in $h \circ p_2$, and the operation op_2 is not decided before op_3 in $h \circ p_3 \circ p_2$.*

Claim 5.8 *If the condition in line 16 is true, then immediately before line 17 the following holds.*
(1.) The next primitive step in the programs of p_1, p_2, and p_3 is to the same memory location.
(2.) The next primitive step in the programs of both p_1 and p_2 is a CAS.
(3.) The expected-value of both the CAS *operations of p_1 and p_2 is the value that appears in the designated address.*
(4.) The new-value of both the CAS *operations is different than the expected-value.*

Proof: By Observation 5.7, in $h \circ p_1 \circ p_3$, the operation op_1 is decided before op_3, while in $h \circ p_3 \circ p_1$, the operation op_1 is not decided before op_3. Thus, in an execution in which p_3 runs solo and completes op_3 immediately after $h \circ p_1 \circ p_3$ it must return a different result than in an execution in which p_3 runs solo and completes op_3 immediately after $h \circ p_3 \circ p_1$ (because each operation by p_1 changes the return value of op_3). Thus, $h \circ p_3 \circ p_1$ and $h \circ p_1 \circ p_3$ must be distinguishable, and thus the next primitive step in the programs of both p_1 and p_3 must be to the same memory location. Similarly, the next primitive step in the programs of both p_2 and p_3 must be to the same memory location, and (1) is proved.

As mentioned, op_3's result is different if p_3 completes op_3 solo immediately after $h \circ p_1$ than op_3's result if p_3 completes op_3 solo immediately after $h \circ p_3 \circ p_1$. Thus, the next primitive step by the program of p_1 cannot be a READ, otherwise the two executions will be indistinguishable by p_3. Similarly, the next primitive step of p_1 cannot be a CAS that does not change the shared memory (i.e., a CAS in which the expected-value is different from the value in the target address, or a CAS in which the expected-value and the new-value are the same.) The symmetric argument for p_2 demonstrates that the next step in the program of p_2 cannot be a READ or a CAS that does not change the shared memory as well.

Thus, the next steps of both p_1 and p_2 are either a WRITE, or a CAS that satisfies (3) and (4). It remains to show the next steps are not a WRITE. Assume by way of contradiction that the next step by p_1 is a WRITE. Thus, $h \circ p_2 \circ p_1$ is indistinguishable from $h \circ p_1$ to all processes excluding p_2. Assume that after either one of these two histories, p_1 runs solo and completes the execution of op_1, and immediately afterwards, p_3 runs solo and completes the execution of op_3. If p_2 executes the next step following h, then op_3 should return a result consistent with an execution in which both op_1 and op_2 are already completed. In the other case, in which the step following h is taken by p_1, op_3 should return a result consistent with an execution in which op_1 is completed and op_2 is not. Since both of these results are different, but the histories are indistinguishable to p_3, we reach a contradiction. Thus, in h, the next step by p_1 is not a WRITE, and similarly, the next step by p_2 is also not a WRITE.

Claim 5.8 immediately implies:

Corollary 5.9 *If the condition in line 16 is true, then the primitive step* p_2 *takes in line 17 is a successful* CAS, *and the primitive step* p_1 *takes in line 18 is a failed* CAS.

Claim 5.10 *If the condition in line 16 is true, then immediately after line 18, the operation* op_1 *is not decided before any operation of* p_3.

Proof: Immediately before line 16, the operation op_1 is not decided before any operation of p_3 by claim 5.6. In a help-free implementation such as S, an operation can only be decided before another operation following a step of its owner process. Following this rule, the only step which could potentially decide op_1 before any operation of p_3 is the step p_1 takes at line 18. By Corollary 5.9, this step is a failed CAS. Thus the state before this step and after this step are indistinguishable to all processes excluding p_1. Assume by way of contradiction that this failed CAS decides op_1 to be before an operation op of p_3. If p_3 is run solo right before the failed CAS of p_1, and this run is continued until op is completed, the result of op should be consistent with op_1 not yet executed (since op_1 cannot be decided before op in a help-free implementation during a solo execution of p_3); If p_3 is run solo right after the failed CAS of p_1, and this run is continued until op is completed, the result of op should be consistent with op_1 already executed. These two scenarios are indistinguishable by p_3, yet the results are different according to the semantics of the specification and the respective programs, yielding a contradiction.

Corollary 5.11 *If the condition in line 16 is true, then immediately after line 18, the operation* op_1 *is not yet completed.*

Proof: Immediately after line 18, the operation op_1 is not decided before any operation of p_3 (Claim 5.10). Were op_1 completed, then by Observation 3.4, it must have also been decided before all future operations of p_3 that have not started yet.

Claim 5.12 *If the condition in line 16 is false, then immediately after line 25, the order between op_j and op_3 is not yet decided. Furthermore, the order between op_j and any of the future operation by p_3 is not decided as well.*

Proof: Immediately before line 14 the order between op_j and op_3, or between op_j and any future operation of p_3 is not yet decided (Claim 5.2). In lines 14–15, p_3 is the only process to advance. By the condition in line 14, op_3 is not decided before neither op_1 or op_2 during the execution of the second inner loop. Furthermore, by the condition in line 16, and by the definition of op_j, op_3 is not decided before op_j after line 24. p_j did not make any step since line 14, and thus op_j cannot be decided before op_3 or any future operation of p_3.

Observation 5.13 *If the condition in line 16 is false, then immediately after line 25, op_j is not yet completed.*

The above is true because op_j did not execute a step since line 14, and was not completed at the time (Claim 5.3).

Claim 5.14 *If the condition in line 16 is false, then immediately after line 25, op_k is not decided before any operation of p_3.*

Proof: According to the condition of line 16 and the definition of k, after line 25 op_k is not decided before op_3. Immediately after line 25, the order between op_3 and op_j is not yet decided (Claim 5.12). Thus, op_k is not decided before op_j (because op_j may still be before op_3, which in turn may still be before op_k). Assume by contradiction that after line 25, op_k is decided before some operation op of p_3. Note that op_j is not decided before op at this point (Claim 5.12). Let p_3 run solo until op is completed.

We claim that after such a run, op is decided before op_j. It is already assumed (contradictively) that op_k is decided before op; no future operations of p_k (operations not yet started) can be decided before the already completed op (Observation 3.4); all operations of p_j before op were completed before op has begun, and are thus before it.

Thus, for every operation $O \neq op_j$ the order between O and op is already decided. According to the semantics of the specification, op returns a different result if op_j is before op than if op_j is after op, given that the relative order between op and all other operations is fixed. Consequently, once op is completed, the order between op_j and op must also be decided. However, op_j cannot be decided before op: it was not decided before op immediately after line 25 and p_j has not taken a step since. The only remaining possibility is that op is decided before op_j.

If op is indeed decided before op_j then by transitivity op_k is decided before op_j. However, since p_3 is not the owner of op_k, then a solo execution of p_3 cannot decide op_k to be before op_j in the help-free S, yielding a contradiction.

Corollary 5.15 *If the condition in line 16 is false, then immediately after line 25, op_k is not yet completed.*

Proof: Immediately after line 25, the operation op_k is not decided before any operation of p_3 (Claim 5.14). Were op_k completed, then by Observation 3.4 it must have been decided before all future operations of p_3 that have not started yet.

Claim 5.16 *Immediately after exiting the loop of lines 26–27, it holds that 1) the operation p_3 has completed operation op_3 and has not yet started the next operation, 2) the operation op_1 has not yet completed, 3) the order between op_1 and any future operation of p_3 is not yet decided, and 4) the order between the first uncompleted operation of p_2 and any future operation of p_3 is not yet decided.*

Proof: By Claim 5.4, op_3 is not completed in lines 14–15. In lines 16–25, p_3 takes at most one step (line 24), because op_3 is not completed in lines 14–15, the step must be a step of op_3, and not of the next operation of p_3. The code of lines 26–27 ensures p_3 will complete op_3 if it is not yet completed, but will not start the next operation, guaranteeing (1).

Now, divide into two cases. If the condition in line 16 is true, then op_2 is completed in lines 19–20. Thus, the first uncompleted operation of p_2 has not yet begun. The order between two operations that have not yet begun cannot be decided (Observation 3.4), and we get (4). The operation op_1 has not yet completed by Corollary 5.11, giving (2). The operation op_1 was not decided before any operation of p_3 after line 18 (Claim 5.10). Process p_1 did not take another step since line 18 and S is a help-free implementation, thus op_1 is not decided before any operation of p_3. Any future operation of p_3 cannot be decided before op_1 by Observation 3.4, and thus we get (3).

If the condition in line 16 is false, then op_1 and op_2 are op_j and op_k (not necessarily in that order). Immediately after line 25, op_k is not decided before any operation of p_3 (Claim 5.14), op_k is not completed (Corollary 5.15), op_j is not decided before any operation of p_3 (Claim 5.12), and op_j is not completed (Observation 5.13)). In lines 26–27 only p_3 may progress, thus both op_1 and op_2 cannot be completed (guaranteeing (2)), and cannot be decided before any other operation since S is help-free (guaranteeing (3) and (4)).

Claim 5.17 *Every iteration of the main loop (lines 2–27) is finite.*

Proof: In every iteration of the main loop, the executions of the first inner loop (lines 6–13) and second inner loop (lines 14–15) are finite (Claim 5.5). The other two inner loops (lines 19–20 and 26–27) run a single process exclusively until it completes its operation, which always takes a finite number of execution steps in a lock-free algorithm. Thus, each execution of an inner loop is finite, as every iteration of the main loop is finite.

Claim 5.18 *S is not wait-free.*

Proof: By Claim 5.17, each iteration of the main loop is finite. It follows that when history h is constructed following the algorithm in Figure 2 the main loop is run infinitely many times. Thus, we consider two cases. The first case is that the condition in line 16 is true only a finite number of times (in a finite number of iterations of the main loop). In this case, We consider the part of history h created since after the last iteration in which the condition in line 16 is true. If the condition is never true, we consider the entire history h. In this part of the history, neither p_1 nor p_2 complete any operation: in each iteration these operations are not completed until after line 25 (Corollary 5.15, Observation 5.13), and only p_3 makes progress in lines 26–27. On the other hand, in each iteration at least one of p_1 and p_2 takes at least one step - in line 25. This contradicts wait-freedom.

The second case is that the condition in line 16 is true infinitely many times. In this case, op_1 is never completed (Claim 5.16 (2)), yet p_1 takes infinitely many steps: each time the condition in line 16 is true, p_1 takes a step in line 18, also contradicting wait-freedom.

Since the assumptions on S were that it is linearizable, help-free, and lock-free, we can rephrase Claim 5.18 as follows.

Theorem 5.19 *A wait-free linearizable single-scanner snapshot implementation cannot be help-free.*

5.1 From Single Scanner Snapshot to Global View Types

The first natural observation is that if a wait-free linearizable single-scanner snapshot cannot be implemented without help, then this conclusion holds for more general snapshot variants as well, such as the multiple-scanner snapshot object, or simply the snapshot object. However, we can generalize the result further. The proof relies on the fact that for every SCAN, its result changes if it is linearized before op_1 and op_2, compared to when it is linearized after the first of op_1 and op_2, and compared to when it is linearized after both.

In what follows, we generalize this result to global view types. Similarly to the proof above, we think of a single operation *op* (similar to op_1 of p_1), an infinite sequence of operations Modifiers (similar to the infinite UPDATE sequence of op_2, and an infinite sequence of operations Views (similar to the infinite SCAN sequence of p_3).

Next, a certain property that holds for every *modifier* and every *view* operation is needed. Specifically, this property states that the view returns a different result if another (either modifier or the *op* operation) is added before it, and yet a different result if both the modifier and *op* are added before it. For this purpose, we define three sets of sequential histories for each pair of modifier and view. Set_0 is histories in which the view is after the specified modifier, but not after any other modifier, and not after *op* either. Set_1 is histories in which either *op* or one more modifier is before the view, and Set_2 is the histories in which both the one more modifier and *op* are before the view. The definition follows.

Definition 5.20 (Modifiers-Viewers Sets.) *Given a single operation denoted op, an infinite sequence of operations denoted Modifiers, and an infinite sequence of operations denoted Views, for every pair of integers $i \geq 0$ and $j \geq 1$ we define the following three* modifier-i-view-j *sets.*

Set_0 *is the set of all sequential histories h that include the first i Modifiers operations in their relative order, include the first j Views operations in their relative order, include no other operation and the last operation in h is in Views.*

Set_1 *is the set of all sequential histories h that include the first i Modifiers operations in their relative order, include the first j Views operations in their relative order, include op or include the $(i+1)$-st operation of Modifiers somewhere after the first i operations of Modifiers but not both, include no other operations, and the last operation in h is in Views.*

Set_2 *is the set of all sequential histories h that include the first $i+1$ operations of Modifiers in their relative order, include the first j operations of Views in their relative order, include op, include no other operations, and the last operation in h is in Views.*

The interests we have in these sets relies in the result of the last (view) operation in each history. Specifically, for our proof to hold, if two histories h and h' belong to two different modifier-i-view-j set, then the results of their last operation should be different. We use the following definition to help formalize this.

Definition 5.21 (Modifiers-Viewers Result Sets.) *Given a single operation denoted op, an infinite sequence of operations denoted Modifiers, and an infinite sequence of operations denoted Views, for every pair of integers $i \geq 0$ and $j \geq 1$ we define the following three* modifier-i-view-j-results *sets as follows:*
$RS_i = \{r | r \text{ is the returned value of the last (view) operation in a history } h \in Set_i\}$

Definition 5.22 (Global View Types.) *A type t, for which there exists an operation op, an infinite sequence of operations Modifiers and an infinite sequence of operations Views, such that for every pair of integers $i \geq 0$ and $j \geq 1$ the three modifier-i-view-j-results sets are disjoint sets, is called a* Global View Type.

Using this definition, Theorem 5.19 is generalized as follows:

Theorem 5.23 *A global view type has no linearizable, wait-free, help-free implementation.*

Both snapshot objects and increment objects are such types.[5] Another interesting type is the fetch&increment type, which is sometimes used as a primitive. This type supports only a single operation, which returns the previous integer value and increments it by one. In the type of the fetch&increment, op, the Modifiers sequence, and the Views sequence, all consists only fetch&increment operations. Its easy to see that for every pair i and j, the three results sets are disjoint, because each set contains histories with a different number of operations. Finally, the fetch-and-cons object, used in [17], is another example of a type that satisfies the condition in theorem 5.23.

6 Max Registers

In this section we turn our attention to systems that support only the READ and WRITE primitives, (without the CAS primitive). We prove that for such systems, help is often required even to enable lock-freedom. We prove this for the *max-register* type [3]. A max-register type supports two operations, WRITEMAX and READMAX. A WRITEMAX operation receives as an input a non-negative integer, and has no output result. A READMAX operation returns the largest value written so far, or 0, if no WRITEMAX operations were executed prior to the READMAX. If the CAS primitive is allowed, then there is a help-free wait-free max-register implementation. (See Subsection 7.2.)

Assume by way of contradiction that M is a linearizable, help-free, lock-free max-register implementation. Consider a system of five processes, p_1, p_2, p_3, p_4, and p_5. The programs of processes p_1, p_2, and p_3 all consists of a single operation, which is WRITEMAX(1), WRITEMAX(2), and WRITEMAX(3) respectively. The programs of processes p_4 and p_5 are both a single READMAX operation. We denote the operations of p_1, p_2, and p_3 by W_1, W_2, and W_3 respectively. We denote the operations of p_4 and p_5 by R_1 and R_2 respectively.

In the proof we build a history h, such that processes p_1, p_2, p_3, p_4, and p_5 follow their respective programs. We show this yields a contradiction to help-freedom. The algorithm for constructing this history is depicted in Figure 3. Processes p_1, p_2 are scheduled to run their programs as long as their operations are not decided before R_1. Process p_4 is scheduled to run its program as long is it can make a step without deciding its operation before W_1, or alternatively, without deciding its operation before W_2. We prove that during the execution of the main loop, no operation is ever decided before any other operation. We also prove that the execution of the main loop must be finite.

Afterwards, the execution of M is in a critical point. If p_1 were to take a step, then W_1 will be decided before R_1; if p_2 were to take a step, then W_2 will be decided before R_1; and if p_4 were to take a step, then R_1 will be decided before both W_1 and W_2. We will prove using indistinguishability arguments that this yields a contradiction.

Claim 6.1 *During the execution of the main loop (lines 2–15), no operation is decided before any other operation.*

Proof: The proof is by induction on the iteration number of the main loop. The induction hypothesis for iteration i is that no operation is decided before another operation during the execution of the first $i-1$ iterations. For the first iteration this is trivial. We assume that the hypothesis is correct for iteration i, and prove that it holds for iteration $i+1$ as well. That is, we prove that given that no operation is decided in the first $i-1$ iterations, no operation is decided in iteration i.

[5] An increment object supports two operations, INCREMENT and GET.

```
1:  h = ε;
2:  while (true)                                          ▷ main loop
3:     if W_1 is not decided before R_1 in h ∘ p_1
4:        h = h ∘ p_1;
5:        continue;                                       ▷ goto line 2
6:     if W_2 is not decided before R_1 in h ∘ p_2
7:        h = h ∘ p_2;
8:        continue;                                       ▷ goto line 2
9:     if R_1 is not decided before W_1 in h ∘ p_4
10:       h = h ∘ p_4;
11:       continue;                                       ▷ goto line 2
12:    if R_1 is not decided before W_2 in h ∘ p_4
13:       h = h ∘ p_4;
14:       continue;                                       ▷ goto line 2
15:    break;                                             ▷ goto line 16
16:    Contradiction          ▷ reaching this line immediately yields contradiction.
```

Figure 3: The algorithm for constructing the history in the proof of Theorem 6.6.

Before the execution of the main loop, no operation is decided before any other operation because h is empty (Observation 3.4). By the induction hypothesis, no operation was decided during the first $i-1$ iterations of the main loop. It follows that if an operation is decided before another operation during iteration i of the main loop, then it must be the first time any operation is decided before a different operation in h. If this indeed happens, then it must be in one of the lines: 4,7,10, or 13. We go over them one by one, and prove that an execution of none of them can be the first time an operation is decided before a different operation.

Assume by way of contradiction that the execution of line 4 in iteration i is the first time in which an operation is decided before a different operation. Because M is a help-free algorithm, the operation that is decided before a different operation must be W_1. Since W_1 is now decided before a different operation, and since W_3 is not yet decided, then in particular, W_1 must be decided before W_3 (Claim 3.5).

At this point, let p_3 run solo until completing W_3. We consider two cases. The first case is that after this run of p_3, W_3 is decided before R_1. If this is the case, then by transitivity, W_1 must also be decided before R_1. However, in a help-free algorithm W_1 cannot be decided before R_1 during a solo execution of p_3. It follows that W_1 was already decided before R_1 prior to this solo execution. Thus, W_1 must be decided before R_1 in the execution of line 4. But this contradicts the condition in line 3, and thus the first case is impossible.

The second case is that after the solo run of p_3 which completes W_3, W_3 is not decided before R_1. If W_3 is not decided before R_1 after the completion of W_3, then it follows that W_3 can never be decided before R_1 in a help-free algorithm (because any such future decision cannot be inside the execution of W_3, and will thus be help.) Thus, R_1 cannot possibly return any value ≥ 3: returning such a value would indicate it is after W_3 (because W_3 is decided to be the first operation that writes a value ≥ 3, as no other operation that writes a value ≥ 3 has even started, and W_3 is already completed). But if R_1 cannot possibly return a value ≥ 3, then R_1 is decided before W_3. However, R_1 cannot be decided before another operation during the execution of line 4 or during the solo execution of p_3 in a help-free algorithm. Since we assumed that line 4 is the first time any operation is decided before any other operation, this yields a contradiction, making the second case impossible as well.

Thus, we have established that line 4 in iteration i of the main-loop cannot decide any operation before any other operation. The argument for line 7 is similar. We move on to consider line 10.

Assume by way of contradiction that the execution of line 10 in iteration i is the first time in which an operation is decided before a different operation. Because M is a help-free algorithm, the operation that is decided before a different operation must be R_1. Since R_1 is now decided before a different operation, and since R_2 is not yet decided, then in particular, R_1 must be decided before R_2 (Claim 3.5).

At this point, let p_5 run solo until completing R_2. We consider two cases. The first case is that R_2 returns 0. If this is the case, then R_2 is decided before both W_1 and W_2, and by transitivity, R_1 is decided before W_1 and W_2 as well. However, R_1 cannot be decided before W_1 in a help-free algorithm during a solo execution of p_5. It follows that R_1 was already decided before W_1 before this solo execution. Thus, R_1 must be decided before W_1 in the execution of line 10. But this contradicts the condition in line 9, and thus the first case is impossible.

The second case is that R_2 returns a value greater than 0. In such a case, either W_1 or W_2 must be decided before R_2 (depending on the value returned). But both W_1 and W_2 cannot be decided before R_2 during the solo execution of p_5, or during the execution of line 10, in a help-free algorithm. Since we assumed line 10 is the first time any operation is decided before any other operation, then this yields a contradiction, making the second case impossible as well.

Thus, we have established that line 10 in iteration i of the main-loop cannot decide any operation before any other operation. The argument for line 13 is similar, and the proof is complete.

Corollary 6.2 *No operation is completed during the execution of the main loop (lines 2–15).*

Proof: By Claim 6.1, no operation is decided before any other operation during the execution of the main loop. By Observation 3.4, an operation that is already completed must be decided before all operations that have not yet started. Since there are operations that are never started (W_3, R_2), but no operation is decided before any operation, then no operation can be completed during the execution of the main loop.

Corollary 6.3 *The execution of the main loop (lines 2–15) is finite.*

Proof: In each iteration of the main loop excluding the last one, a process takes a step in M. However, during the execution the main loop no operation is completed. (Corollary 6.2.) Since M is a lock-free implementation then this cannot continue infinitely, and thus the execution of the main loop is finite.

Observation 6.4 *Immediately before line 16 the order of any two operations is not yet decided. Furthermore, immediately before line 16, it holds that 1) in $h \circ p_1$, the operation W_1 is decided before R_1, 2) in $h \circ p_2$, the operation W_2 is decided before R_1, and 3) in $h \circ p_4$, the operation R_1 is decided before both W_1 and W_2.*

From Claim 6.1 the order between any two operations is not decided immediately before line 16.
From observing the code, the main loop exits and line 16 is reached only if (1), (2), and (3) hold.

Claim 6.5 *Reaching line 16 yields contradiction.*

Proof: By Observation 6.4, in $h \circ p_1 \circ p_4$, the operation W_1 is decided before R_1, and in $h \circ p_4 \circ p_1$, the operation R_1 is decided before both W_1 and W_2. It follows that $h \circ p_1 \circ p_4$ and $h \circ p_4 \circ p_1$ must be distinguishable to process p_4, since if p_4 continues to run solo after $h \circ p_4 \circ p_1$ then R_1 must return 0, and if p_1 runs solo after $h \circ p_1 \circ p_4$ then R_1 must return at least 1. It follows that the next steps of both p_1 and p_4 must be to the same memory address. Furthermore, to enable distinguishability by p_4, the next step by p_1 must be a WRITE, and the next step by p_4 must be a READ.

For similar reasons, the next step of p_2 must also be a WRITE to the same memory address. Thus, the next step by both p_1 and p_2 is a WRITE to the same location. Thus, $h \circ p_2 \circ p_1$ and $h \circ p_1$ are indistinguishable to p_4. Since in $h \circ p_2 \circ p_1$ the operation W_2 is decided before R_1, a solo execution by p_4 starting from that point until R_1 is completed must cause R_1 to return 2. Since this is indistinguishable to p_4 from $h \circ p_1$, then a solo execution of p_4 immediately after $h \circ p_1$ must also return 2. However, this would imply W_2 is decided before R_1. But W_2 is not decided before R_1 in h (Observation 6.4), and cannot be decided before it during a step of p_1 or during the solo execution of p_4 in a help-free algorithm, yielding a contradiction.

Theorem 6.6 *A lock-free implementation of a max-register using only* READ *and* WRITE *primitives cannot be help-free.*

Proof: We assumed a lock-free help-free implementation of a max-register using only READ and WRITE primitives. However, while examining the algorithm for constructing history h depicted in Figure 3, we reached the conclusion that the main-loop execution must be finite (Corollary 6.3), but also the conclusion that line 16 can never be reached (Claim 6.5). This yields contradiction, and proves the Theorem.

7 Types that Do Not Require Help

In this section, we establish that some types can be implemented in a wait-free manner without using help. Loosely speaking, if the type operations dependency is weak enough then no help is required. As a trivial example, consider the *vacuous type*. A vacuous object supports only one operation, NO-OP, which receives no input parameters and returns no output parameters (void). Thus, the result of a NO-OP does not depend on the execution of any previous operations. Consequently, there is no operations dependency at all in the vacuous type. It can trivially be implemented by simply returning void without executing any computation steps, and without employing help.

7.1 A Help-Free Wait-Free Set

As a more interesting example, consider the *set* type of a finite domain. The set type supports three operations, INSERT, DELETE, and CONTAINS. Each of the operations receives a single input parameter which is a key in the set domain, and returns a boolean value. An INSERT operation adds the given key to the set and returns true if the key is not already in the set, otherwise it does nothing and returns false. A DELETE operation removes a key from the set and returns true if the key is present in the set, otherwise it does nothing and returns false. A CONTAINS operation returns true if and only if the input key exists in the set.

Consider the following wait-free help-free set implementation given in Figure 4. The implementation uses an array with a bit for every key in the set domain. Initially, all bits are set to zero, and the set is empty. To insert a key to the set, a process performs a CAS operation that changes the bit from zero to one. If the CAS succeeds, the process returns true. If the CAS fails, that means that the key is already in the set, and the process returns false. Deletion is executed symmetrically by CASing from one to zero, and contains reads the appropriate bit and returns true if and only if it is set to one.

In this set algorithm, it is easy to specify the linearization point of each operation. In fact, every operation consists of only a single computation step, which is the linearization point of that operation. For any type, an obstruction-free implementation in which the linearization point of every operation can be specified as a step in the execution of *the same* operation is help-free.

The function f that proves such an implementation is help-free is derived naturally from the linearization points. For each given history, the operations are ordered according to the order of the execution of their

```
bool insert(int key) {
    bool result = CAS(A[key],0,1);                ▷ linearization point
    return result; }
bool delete (int key) {
    bool result = CAS(A[key],1,0);                ▷ linearization point
    return result; }
bool contains (int key) {
    bool result = (A[key] == 1);                  ▷ linearization point
    return result; }
```

Figure 4: A help-free wait-free set implementation

linearization points. Consider a type T, an obstruction-free implementation of it O, and the corresponding set of histories H. Assume the code of O specifies the linearization point of each operation at the execution of a specific computation step of the same operation. Let f be the linearization function derived from this specification.

Claim 7.1 *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 the linearization point of op_1.*

As a direct result, γ is executed by the owner of op_1, and thus O is help-free.

Proof: First, we observe that op_1 is not yet linearized in h. If it were, then the order between op_1 and op_2 would have already been decided: were op_2 linearized before op_1 then op_2 would have been decided before op_1, and were op_1 linearized before op_2 or op_1 is linearized and op_2 not, then op_1 would have been decided before op_2. Thus, op_1 cannot be linearized in h.

Second, we observe that op_1 is linearized in $h \circ \gamma$. Were it not, then a solo execution of the owner of op_2 until the linearization of op_2 would have linearized op_2 before op_1, contradicting the assumption that op_1 is decided before op_2 in $h \circ \gamma$.

7.2 A Help-Free Wait-Free Max Register

In Section 6 we proved that a lock-free max register cannot be help-free if only READS and WRITES are available. In this subsection we show that a help-free wait-free max register is possible when using the CAS primitive. The implementation uses a shared integer, denoted `value`, initialized to zero. This integer holds the current max value. The implementation is given in Figure 5.

A WRITEMAX operation first reads the shared integer `value`. If it is greater than or equal to the input key, then the operation simply returns. Otherwise it tries by a CAS to replace the old (smaller) value with the operation's input key. If the CAS succeeds, the operation returns. Otherwise the operation starts again from the beginning. This implementation is wait-free because each time the CAS fails, the shared `value` grows by at least one. Thus, a WRITEMAX(x) operation is guaranteed to return after a maximum of x iterations. A READMAX operation simply reads the `value` and returns it.

Help-Freedom is proved similarly to the wait-free help-free set, using Claim 7.1. In the given max register implementation, the linearization point of every operation can be specified as a step in the execution of *the same* operation, and thus it is help-free. The linearization point of a WRITEMAX operation is always its last computation step. This is either reading the `value` variable (if the read value is greater than the input key), or the CAS (if the CAS succeeds). The linearization point of a READMAX is reading the `value`.

```
void WriteMax(int key) {
    while(true) {
        int local = value;                          ▷ linearization point if value ≥ key
        if (local ≥ key)
            return;
        if (CAS(value, local, key));                ▷ linearization point if the CAS succeeds
            return;
} }
int ReadMax() {
    int result = value;                             ▷ linearization point
    return result;
}
```

Figure 5: A help-free wait-free max register implementation

8 A Universality of Fetch-And-Cons

A fetch-and-cons object allows a process to atomically add (con) an item to the beginning of a list and return the items following it. In this section, we show that fetch-and-cons is universal with respect to help-free wait-free linearizable objects. That is, given a help-free wait-free atomic fetch-and-cons primitive, one can implement any type in a linearizable wait-free help-free manner. Not surprisingly for a universal object, both Theorems 4.18 and 5.23 hold for fetch-and-cons and show it cannot be implemented in a help-free wait-free manner. Before demonstrating the universality of fetch-and-cons, we shortly discuss the strength of different primitives when it comes to overcoming indistinguishability problems.

Consider two processes, p_1 and p_2, at a certain point in an execution. Consider only their immediate next computation step. With this regard, there are five possible states: 1) neither have yet taken its next step, 2) p_1 has taken its next step and p_2 has not, 3) p_2 has taken its next step and p_1 has not, 4) p_1 has taken its next step, and afterwards p_2 has taken its next step, and 5) p_2 has taken its next step, and afterwards p_1 has taken its next step. Different primitives can be measured by their ability to support distinguishability between each of these five possibilities. Perfect distinguishability allows each process in the system to know exactly which one of the five scenarios occurred.

Using such a metric, we can state that a system supporting only READ and WRITE is weaker than a system that also supports CAS. When both p_1 and p_2 are attempting a CAS at the same memory location, it is possible for every process in the system to distinguish between (4) and (5), while also distinguishing between (3) and (4). This is impossible when using only READ and WRITE. Still, a CAS is not perfect: for example, it is still impossible to distinguish between (2), (3) and (4) at the same time.

FETCH&ADD adds more strength to the system. When both p_1 and p_2 execute FECTH&ADD, in which they add different values to the same location, it is possible for every process in the system to distinguish between (1), (2), (3), and (4). In fact, FETCH&ADD is almost perfect: its only weakness is that it does not allow processes other than p_1 and p_2 to distinguish between (4) and (5). By contrast, fetch-and-cons is perfect: it allows every process in the system to distinguish between all five possibilities. Intuitively, this is the source of its strength.

To show that fetch-and-cons is indeed universal, we use a known wait-free reduction from any sequential object to fetch-and-cons, described in detail in [17]. We claim that the reduction is help-free. In essence, each process executes every operation in two parts. First, the process calls fetch-and-cons to add the description of the operation (such as ENQUEUE(2)) to the head of the list, and gets all the operations that preceded it. This fetch-and-cons is the linearization point of the operation.

Second, the process computes the results of its operation by examining all the operations from the beginning of the execution, and thus determining the "state" prior to its own operation and the appropriate result. Note that since every operation is linearized in its own fetch-and-cons step, then this reduction is help-free by Claim 7.1.

9 Discussion

This paper studies the fundamental notion of help for wait-free concurrent algorithms. It formalizes the notion, and presents conditions under which concurrent data structures must use help to obtain wait-freedom.

We view our contribution as a lower-bound type of result, which sheds light on a key element that implementations of certain object types must contain. As such, we hope it will have a significant impact on both research and design of concurrent data structures. First, we believe it can lead to modularity in designs of implementations that are shown to require a helping mechanism in order to be wait-free, by allowing to pinpoint the place where help occurs.

Second, we ask whether our definition of help can be improved in any sense, and expect this to be an important line of further research. We think that our proposed definition is a good one, but there exist other possible definitions as well. An open question is how various formalizations of this notion relate to each other. Another important open problem is to find a definition for the other notion of help, as we distinguish in the introduction. Such a definition should capture the mechanisms that allow a process to set the ground for its own operation by possibly assisting another operation, for the sole purpose of completing its own operation. In this paper we do not refer to the latter as help, as captured by our definition.

An additional open problem is the further characterizations of families of data structures that require help to obtain wait-freedom. For example, we conjecture that *perturbable objects* [18] cannot have wait-free help-free implementations when using only READ and WRITE primitives, but the proof would need to substantially extend our arguments for the max register type.

Acknowledgements: The authors thank Nir Shavit for intriguing discussions, and the anonymous referees of a previous version of the paper for helpful comments.The first author is supported in part by ISF Grant 1696/14. The second and third authors are supported in part by ISF Grant 274/14.

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993.

[2] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? In *Symposium on Theory of Computing, (STOC)*, pages 714–723, 2014.

[3] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, March 2012.

[4] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

[5] Greg Barnes. A method for implementing lock-free shared-data structures. In *SPAA*, pages 261–270, 1993.

[6] Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 115–124, 2012.

[7] Faith Ellen, Danny Hendler, and Nir Shavit. On the inherent sequentiality of concurrent objects. *SIAM J. Comput.*, 41(3):519–536, 2012.

[8] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *SPAA*, pages 325–334, 2011.

[9] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free algorithms can be practically wait-free. In *19th International Symposium on Distributed Computing (DISC)*, pages 78–92, 2005.

[10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[11] Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 373–382, 2011.

[12] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 197–206, 1990.

[13] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[14] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[15] Maurice Herlihy and Nir Shavit. On the nature of progress. In *Proceedings of the 15th International Conference on Principles of Distributed Systems (OPODIS)*, pages 313–328, 2011.

[16] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[17] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 276–290, 1988.

[18] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, 2000.

[19] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *PPOPP*, pages 223–234, 2011.

[20] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *PPOPP*, pages 141–150, 2012.

[21] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, August 1974.

[22] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275, 1996.

[23] S. A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 159–175, 1989.

[24] Eric Ruppert. Determining consensus numbers. *SIAM J. Comput.*, 30(4):1156–1168, 2000.

[25] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *OPODIS*, 2012.

[26] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 357–368, 2014.