History-Independent Concurrent Objects

Hagit Attiya
Technion
Haifa, Israel
hagit@cs.technion.ac.il

Michael A. Bender
Stony Brook University
Stony Brook, NY, USA
bender@cs.stonybrook.edu

Martín Farach-Colton
New York University
New York, NY, USA
martin@farach-colton.com

Rotem Oshman
Tel-Aviv University
Tel-Aviv, Israel
roshman@tau.ac.il

Noa Schiller
Tel-Aviv University
Tel-Aviv, Israel
noaschiller@mail.tau.ac.il

ABSTRACT

A data structure is called *history independent* if its internal memory representation does not reveal the history of operations applied to it, only its current state. In this paper we study history independence for concurrent data structures, and establish foundational possibility and impossibility results. We show that a large class of concurrent objects cannot be implemented from smaller base objects in a manner that is both wait-free and history independent; but if we settle for either lock-freedom instead of wait-freedom or for a weak notion of history independence, then at least one object in the class, multi-valued single-reader single-writer registers, *can* be implemented from smaller base objects, binary registers.

On the other hand, using large base objects, we give a strong possibility result in the form of a universal construction: an object with s possible states can be implemented in a wait-free, history-independent manner from compare-and-swap base objects that each have $O(s + 2^n)$ possible memory states, where n is the number of processes in the system.

CCS CONCEPTS

• **Theory of computation → Distributed computing models**; **Data structures design and analysis**; **Concurrent algorithms**; • **Computing methodologies → Concurrent algorithms**.

KEYWORDS

state-quiescent history independence, multi-valued register, queue, universal implementation

ACM Reference Format:
Hagit Attiya, Michael A. Bender, Martín Farach-Colton, Rotem Oshman, and Noa Schiller. 2024. History-Independent Concurrent Objects. In *ACM Symposium on Principles of Distributed Computing (PODC '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3662158.3662814

PODC '24, June 17–21, 2024, Nantes, France

ACM ISBN 979-8-4007-0668-4/24/06
https://doi.org/10.1145/3662158.3662814

1 INTRODUCTION

A data structure is said to be *history independent* (HI) if its internal representation reveals only the current state of the data structure, and does not depend on the specific history of operations that led to the current state. For example, in a history-independent implementation of a set, the internal memory representation may (and must, for correctness) reveal the elements that are currently in the set, but it must not reveal elements that were previously inserted and then removed.

The notion of history independence was introduced by Micciancio [32], who showed how to build a search tree with a history-independent structure. Naor and Teague [35] formalized two now-classical notions of history independence: a data structure is *weakly history independent* (WHI) if it leaks no information to an observer who sees the memory representation once, and it is *strongly history independent* (SHI) if it leaks no information even to an observer who sees the memory representation at multiple points in the execution. These notions differ significantly: for example, a set where each item inserted is stored at a freshly-chosen random location in memory may be weakly HI but not strongly HI, because if an item is inserted, removed, and then inserted again, it may be placed in different locations each time it is inserted; an observer who sees the memory after each of the two insertions would know that the item was removed and re-inserted.

History independence has been extensively studied in sequential data structures (see below), and the foundational algorithmic work on history independence has found its way into secure storage systems and voting machines. However, history independence has been studied only peripherally in *concurrent* data structures. This paper initiates a thorough study of history-independent concurrent data structures, and establishes fundamental possibility and impossibility results.

Defining history independence for concurrent objects is non-trivial, because the sequential definition of history independence allows the observer to examine the memory only *in-between* operations—that is, in quiescent states—while a concurrent implementation might *never be* in a quiescent state. One of our main questions is whether it should be permissible for the observer to open the "black box" of a single operation and inspect the memory when the system is not quiescent, and what are the implications of this choice in terms of what can be implemented concurrently in a history independent manner. We focus on a concurrent notion of strong history independence for deterministic data structures, and characterize the boundaries of what can be achieved.

We begin by asking whether a concurrent object A can be implemented out of "smaller" base objects of type B in a manner that is history independent; here, "smaller" means that B has fewer states than A. Our motivating example is the famous implementation of a multi-valued single-writer single-reader wait-free register from Boolean registers [45]. We observe that this implementation is not history independent even in the weakest sense (see Section 4), and in fact, there is a good reason for this: we prove that for a fairly general class of objects, which includes read/write registers, there is no wait-free history independent implementation out of smaller base objects, regardless of the type of the smaller objects. This result holds even if the observer can only inspect the memory when there are *no state-changing operations pending*, but read operations may be ongoing. If the observer can inspect the memory at any point (including while state-changing operations are ongoing), then even a lock-free implementation is impossible. On the other hand, for multi-valued single-writer single-reader registers, there is a wait-free history independent implementation from binary registers, if we restrict the observer's inspections to points where the system is completely quiescent. While our results are stated for deterministic algorithms, our impossibility result also applies to randomized implementations of *reversible objects*, which are objects where every state can be reached from every other state (see Section 7).

Since it is impossible to implement some objects out of smaller objects in a wait-free, history independent manner (except possibly in the weakest, quiescent sense), we turn to concurrent implementations where the base object B is large enough to store the full state of the abstract object A that we want to implement. For this regime we give a strong possibility result, in the form of a *universal implementation* from compare-and-swap (CAS) objects: we show that any object A can be implemented in a wait-free, history independent manner from sufficiently-large CAS objects. Our implementation reveals nothing about past states of the object or operations that completed prior to the invocation of any currently-pending operations; and when no state-changing operation is pending, the state of the memory reflects only the current abstract state of the object. Our implementation uses an extended version of an LL/SC object, inspired by [30, 31], which we implement from atomic CAS.

Additional Related Work. Hartline et al. [24, 25] showed that a data structure with a strongly-connected state graph is strongly history independent (SHI) if and only if each of its state has a unique *canonical* representation (see Proposition 1). We rely on a similar characterization for both possibility and impossibility results.

There is a large body of literature characterizing which data structures can be made history independent without an asymptotic slow down. These results include fast HI constructions for cuckoo hash tables [34], linear-probing hash tables [15, 23], other hash tables [15, 35], trees [1, 32], memory allocators [23, 35], write-once memories [33], priority queues [16], union-find data structures [35], external-memory dictionaries [12, 20–22], file systems [8, 10, 11, 39], cache-oblivious dictionaries [12], order-maintenance data structures [15], packed-memory arrays/list-labeling data structures [12, 13], and geometric data structures [44]. Given the strong connection between history independence and unique representability [24, 25], some earlier data structures can be made history independent, including hashing variants [2, 17], skip lists [37], treaps [5], and other less well-known deterministic data structures with canonical representations [3, 4, 38, 42, 43].

The algorithmic work on history independence has found its way into systems. There are now voting machines [14], file systems [8, 9, 11], databases [10, 36, 40], and other storage systems [18] that support history independence as an essential feature.

To the best of our knowledge, the only prior work to consider history independence in concurrent implementations is by Shun and Blelloch [41]. They implement a concurrent hash table, based on a sequential SHI hash table [15, 35], in which only operations of the same type can be executed concurrently. The implementation of [41] guarantees that if there are no ongoing insert or delete operations, each state of the hash table has a unique canonical representation in memory. This work does not provide a formal definition of history independence for concurrent implementations, and does not support concurrent operations of different types.

Our universal implementation draws inspiration from prior universal implementations, using CAS [26], hardware LL/SC [28], and consensus objects [27]. These implementations are not history independent: the implementation in [27] explicitly keeps tracks of all the operations that have ever been invoked, while the implementations in [26, 28] store information that depends on the sequence of applied operations. Moreover, they use dynamic memory, and allocate new memory every time the state of the object is modified, which risks revealing information about the history of operations. While there is work on sequential history-independent memory allocation (e.g., [35]), to our knowledge, no *concurrent* history-independent memory allocator is known.

Fatourou and Kallimanis [19] give a universal implementation from hardware LL/SC, where the full state of the object, along with additional information, is stored in a single memory cell. Our history-independent universal implementation bears some similarity to [19], but their implementation stores information about completed operations, such as their responses, and is therefore not history independent. Clearing this type of information from memory so as not to reveal completed operations is non-trivial, and we address this in our implementation.

2 PRELIMINARIES

Abstract Objects. An *abstract object* is defined by a set of states and a set of operations, each of which may change the state of the object and return a response. Formally, an abstract object O is a tuple (Q, q_0, O, R, Δ), where Q is the object's set of states, $q_0 \in Q$ is a designated initial state, O is the object's set of operations, R is the object's set of responses and $\Delta : Q \times O \to Q \times R$ is the function specifying the object behavior, known as the *sequential specification* of the object. We assume that the abstract object is deterministic, i.e., the function Δ is deterministic. We also assume that all states in Q are reachable from the initial state q_0.

A *sequential implementation* specifies how each operation should be concretely implemented in memory. The object's state is represented in memory using some *memory representation*, and the implementation specifies how each operation should modify this memory representation when it is applied. In general, a single state $q \in Q$ may have multiple possible memory representations associated with it; for example, if we implement a set (i.e., a dictionary)

using a balanced search tree, then the abstract state of the object consists of the contents of the set, but the layout in memory may also depend on the order of insertions, etc.

Sequential History Independence. A *history-independent* implementation is one where the memory representation of the object reveals only its current abstract state, not the sequence of operations that have led to that state. Where deterministic implementations are concerned, both versions of history independence—weak and strong history independence—coincide, since in the absence of randomness it does not matter whether the adversary examines the memory at one point or or at multiple points in the execution. Because we consider only deterministic implementations in this paper, we do not distinguish between weak and strong history independence, and refer to them simply as *history independence* (HI). (The formal definitions of sequential weak and strong history independence are omitted here; see [35].)

One way to achieve history independence is to ensure that whenever the object is the same abstract state, the memory representation is the same. Implementations that have this property are called *canonical*: every abstract state $q \in Q$ corresponds to exactly one memory representation, $can(q)$, which we call its *canonical memory representation*, and every sequence of operations that leads the object to state q must leave the memory in state $can(q)$. As we said above, for deterministic implementations, weak and strong history independence coincide. Both require the implementation to be canonical: [Consider whether we should still be talking about WHI/SHI at this point, rather than just HI.]

Proposition 1. *For deterministic sequential implementations, WHI and SHI are equivalent to requiring that a unique canonical memory representation is determined for each state at initialization.*

The following example illustrates the concept of history independence and of canonical implementations. Consider an implementation of an insert-only set $S \subseteq [n]$ where the elements of the set are stored in an array of length n, which is initially empty. One possible implementation of an Insert(x) operation is to simply scan the array and store x in the first empty cell, but this implementation is not history independent, because the order of elements in the array reveals the order in which they were inserted. To address this, we could store the elements in the array in *sorted* order. This implementation would indeed be history independent, and the canonical representation $can(S)$ of a given set S would be an array where the elements of S appear in sorted order, with empty cells padding the array up to length n. (A similar idea forms the basis for a HI linear-probing hash table in [35].)

The Asynchronous Shared-Memory Model. We assume the standard model in which n processes, $p_1, \ldots, p_n$, communicate through shared *base objects*. When implementing an abstract object O, it is called the *high-level* object. An implementation of an abstract object O, specifies a program for each process and for each operation in O. Upon receiving an *invocation* of an operation $o \in O$, process p_i takes *steps* according to this program. Each step by process p_i consists of some local computation and a single primitive operation on a base object. After each step, the process may change its local state, and eventually it may return a *response* to the high-level operation.

A *configuration* C specifies the local state of every process and the state of every base object. The *initial configuration* of the system is denoted by C_0, and we assume that it is unique. An *execution* α is an alternating sequence of configurations and steps, starting from the initial configuration C_0. An execution can be finite or infinite. Given two executions α_1, α_2, where α_1 ends at configuration C and α_2 begins at configuration C, we denote by $\alpha_1\alpha_2$ the concatenation of the two executions, and we say that $\alpha_1\alpha_2$ *extends* execution α_1.

The *memory representation* of a configuration C, denoted mem(C), is a vector specifying the state of each base object; this does not include local private redundant, get rid of either local or private variables held by each process, only the shared memory. Formally, if there are m base objects with state spaces $Q_1, \ldots, Q_m$ respectively, then mem(C) is a vector in $Q_1 \times \ldots \times Q_m$, and we denote by mem$(C)[i]$ the state of the i-th base object. For a finite execution α, let mem(α) denote the memory representation in the last configuration in execution α.

In this paper we frequently make use of two types of base objects: the first is a simple *read/write register*, and the second is a *compare-and-swap (CAS)* object. A CAS object X supports the operation CAS(X, old, new), which checks if the current value of the object is *old*, and if so, replaces it by *new* and returns *true*; otherwise, the operation leaves the value unchanged, and returns *false*. We assume that the CAS object supports standard read and write operations. For both read/write registers and CAS objects, the *state* of the object is simply the value stored in it. Note that both registers and CAS objects can also serve as high-level objects. [I would get rid of this sentence, it seems obvious. If we keep it, we should refer specifically to the places in the paper where we think of registers/CAS as high-level objects, and we can say, e.g., "Although we use registers and CAS objects are as base objects, we are also interested in implementing them out of other base objects, such as *smaller* registers and CAS objects; see Section whatever".]

An execution α induces a *history* $H(\alpha)$, consisting only of the invocations and responses of high-level operations. An invocation *matches* a response if they both belong to the same operation. An operation *completes* in H if H includes both the invocation and response of the operation; if H includes the invocation of an operation, but no matching response, then the operation is *pending*. If α ends with a configuration C, and there is no pending operation in α, then C is *quiescent*.

A history H is *sequential* if every invocation is immediately followed by a matching response. For a sequential history H, let state$(H) \in Q$ be the state of the high-level object reached by starting from the initial configuration and applying the sequence of operations invoked (and immediately completed) in H.

Linearizability. A *completion* of history H is a history H' whose prefix is H, and whose suffix includes zero or more responses of pending operations in H. Let comp(H) be the set of all completions of H. A sequential history H' is a *linearization* of an execution α that arises from an implementation of an abstract object O if: (1) H' is a permutation of a history in comp$(H(\alpha))$, (2) H' matches the sequential specification of O, and (3) H' respects the real-time order of non-overlapping operations in $H(\alpha)$. [Real-time order and non-overlapping are not defined, but that's probably fine.] [Use inparaenum if acm format allows, if not check what it does allow.]

An execution α is *linearizable* [29] if it has a linearization, and an implementation of an abstract object is linearizable if all of its executions are linearizable. A *linearization function* h maps an execution α to a sequential history $h(\alpha)$ that is a linearization of α.

Progress Conditions. An implementation is *lock-free* if there is a pending operation, then some operation returns in a finite number of steps. An implementation is *wait-free* if there is a pending operation by process p_i, then this operation returns in a finite number of steps by process p_i. [Preceding paragraph is grammatically incorrect. It is also informal since it doesn't use the terminology of executions, histories, etc. but that's probably fine.]

3 HISTORY INDEPENDENCE FOR CONCURRENT OBJECTS

As we noted in Section 1, when defining history independence for concurrent objects, we must grapple with the fact that a concurrent system might never be in a quiescent state. In Section 5.2 we prove that if we allow the internal memory to be observed at *any* point in the execution, then there is a strong impossibility result ruling out even lock-free implementations of a wide class of objects. This motivates us to consider weaker but more feasible definitions, where the observer may only examine the internal memory at certain points in the execution.

The following definition provides a general framework for defining a notion of history independence that is parameterized by the points where the observer is allowed to access the internal memory; these points are specified through a set of finite executions, and the observer may access the memory representation only at the end of each such finite execution. Informally, the definition requires that at any two points where the observer is allowed to examine the internal memory, if the object is in the same state, then the memory representation must be the same; we use a linearization function to determine what the "state" of the object is at a given point.

Definition 2. *Consider an implementation of an abstract object and let E be a set of finite executions that arise from the implementation. The implementation is* HI with respect to E *if there is a linearization function h such that for any pair of executions $\alpha, \alpha' \in E$ such that $state(h(\alpha)) = state(h(\alpha'))$ we have $mem(\alpha) = mem(\alpha')$.*

[Need to improve the way this definition is written. As is, it appears to define "HI with respect to E" twice. The first time is w.r.t. a given linearization function, but that's not the way it's written, it just says "the implementation is HI" without referring to h. Possible solution: only one definition, "An implementation is HI with respect to E if there exists a linearization function h such that [condition]."]

To prove that an implementation satisfies Definition 2, it suffices to find a linearization function h and a canonical representation $can(\cdot)$, and prove that for every finite execution $\alpha \in E$ that ends with the object in state $q \in Q$ (according to h), the memory is in the canonical memory representation $can(q)$.

The strongest form of history independence that one might ask for is one that allows the observer to examine the memory at any point in the execution:

Definition 3. *An implementation of an abstract object is* perfect HI *if the implementation is HI with respect to the set containing all finite executions of the implementation.*

Perfect HI imposes a very strong requirement on the implementation: intuitively, any two adjacent high-level states must have adjacent canonical memory representations. Formally, we say that the *distance* between two memory representations mem_1, mem_2 is d if there are exactly d indices $i \in [m]$ where $mem_1[i] \neq mem_2[i]$ (recall that m is the number of base objects used in the implementation). The following proposition, proved in the full version of this paper [7], holds for *obstruction-freedom*, a progress guarantee even weaker than lock-freedom and wait-freedom, where operations are only required to complete if the process executing them runs by itself for sufficiently long.

Proposition 4. *In any obstruction-free perfect HI implementation of an abstract object with state space Q, for any $q_1 \neq q_2 \in Q$ such that q_2 is reachable from q_1 in a single operation, the distance between $can(q_1)$ and $can(q_2)$ is at most 1.*

Proposition 8 in Section 5.2 shows that a large class of objects do not admit an implementation that meets the requirement above, and therefore, do not have an obstruction-free implementation that is perfect HI. This motivates us to consider weaker definitions, where the observer may only observe the memory at points that are "somewhat quiescent".

We say that an operation $o \in O$ is *state-changing* if there exist states $q \neq q'$ such that o causes the object to transition from state q to q'. An operation is *read-only*[1] if it is not state-changing. A configuration C is *state-quiescent* if there are no pending state-changing operations in C. Note that a quiescent configuration is also state-quiescent.

Definition 5. *An implementation of an abstract object is* state-quiescent HI *if the implementation is HI with respect to all finite executions ending with a state-quiescent configuration.*

The final definition is the weakest one that we consider, and it allows the observer to examine the memory only when the configuration is fully quiescent:

Definition 6. *An implementation of an abstract object is* quiescent HI *if the implementation is HI with respect to all finite executions ending with a quiescent configuration.*

Figure 1 illustrates the three definitions of history independence using the example of a read/write register; we highlight several different points where an observer is or is not allowed to examine the memory according to each definition.

Clearly, if an implementation of an abstract object [Now we're referring to a specific linearization function. This is not necessarily consistent with the definition, which as I commented before doesn't treat h clearly.] is HI with respect to an execution set E, then it is also HI with respect to any execution set $E' \subseteq E$. Thus, perfect HI implies state-quiescent HI, which in turn implies quiescent HI.

4 MOTIVATING EXAMPLE: MULTI-VALUED REGISTER FROM BINARY REGISTERS

To better understand the notion of history independence and the challenges in achieving it, consider Vidyasankar's wait-free implementation of a single-writer single-reader (SWSR) K-valued register

[1] Read-only and state-changing operations are sometimes referred to in the literature as *trivial* and *nontrivial* operations, respectively.

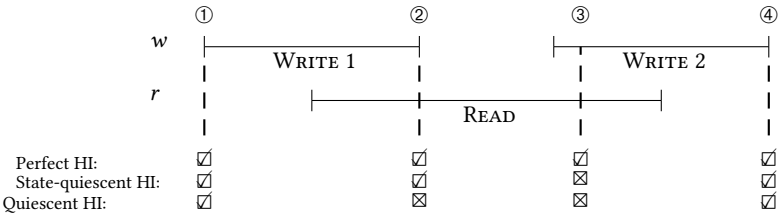

Figure 1: Illustration of the three HI definitions. Perfect HI allows the observer to examine the memory at any point; state-quiescent HI allows inspection only when there is no state-changing operation pending (points 1, 2 and 4); quiescent HI allows inspection only when the configuration is quiescent (points 1 and 4).

(for $K \geq 3$) from binary registers [45] (The original algorithm is for multiple readers, but we assume a single reader here.) The value of the register is represented by a binary array A of size K, and intuitively, the register's value at any given moment is the smallest index $i \in [K]$ such that $A[i] = 1$.[2] In a READ operation, the reader ρ *scans up* to find the smallest index $i \in [K]$ such that $A[i] = 1$, and then *scans down* from i and returns the smallest index $j \leq i$ such that when $A[j]$ was read, its value was 1. (If the reader executes solo, we will have $i = j$, but if there is a concurrent WRITE, we may have $j < i$.) In a WRITE(v) operation, the writer w writes 1 to $A[v]$, then clears the array below index v by writing 0 to all indices $i < v$, starting from index $v - 1$ and proceeding down to index 1.

Since a WRITE(v) operation does not clear values larger than v, the state of the array A leaks information about past values written to the register: e.g., if $K = 3$ and there is a WRITE(2) operation followed by WRITE(1), we will have $A = [1, 1, 0]$, whereas if we have only a WRITE(1), the state will be $A = [1, 0, 0]$. This will happen even in sequential executions, so this implementation is not history independent even in the minimal sense: it does not satisfy the sequential definition of history independence, even if we consider only sequential executions.

One might hope that this can be fixed by having a WRITE(v) operation clear the entire array A, except for $A[v] = 1$, but this would break the wait-freedom of the implementation: if the writer zeroes out all positions in the array except one, we can construct executions where due to overlapping writes, the reader never finds an array position i where $A[i] = 1$, and thus it cannot return a value. In fact, the impossibility result that we prove in the next section (Theorem 11) rules out any wait-free implementation of K-valued registers from binary registers that is history-independent, even if we examine the memory only when no WRITE operation is pending (i.e., wait-free state-quiescent HI implementations).

Nevertheless, if we are willing to settle for lock-freedom instead of wait-freedom, then the approach of clearing out the array after a WRITE can be made to work. We can modify Vidyasankar's algorithm by having a WRITE(v) operation first clear values down from $v - 1$ to 1 (as in the original implementation), and then clear values up from $v + 1$ to K (which it would not do in the original implementation). As a result, when there is no WRITE operation pending, the register has a unique representation: if its value is v, then the array A is 0 everywhere except at v, where we have $A[v] = 1$. Since the reader does not write to memory, this implies that the algorithm is state-quiescent HI. The READ operation is nearly identical to Vidyasankar's algorithm, except that, as we said above, if a READ operation overlaps with multiple WRITEs, it may *not* find a 1 in the array, requiring it to repeat its scan of the array until it finds a value to return. While the WRITE operation remains wait-free, the READ operation is only *lock-free*, as it is only guaranteed to terminate if it eventually runs by itself. The code of the modified algorithm and its proof appear in [7].

There is another way to relax our requirements and circumvent the impossibility result: we could settle for *quiescent* history independence, where the observer is not allowed to inspect the memory representation except when the system is fully quiescent. To do this, we have the reader announce its presence to the writer whenever it begins a read operation, by writing to a special register. The writer, if it sees that the reader might not find a value to return, *helps* it by writing a value that the reader is allowed to return, in an area of shared memory dedicated to this purpose. We must carefully manage the footprints left in memory by both the reader and the writer, to ensure that when all operations complete the memory is left in canonical representation, but at the same time, that the reader is never left hanging without a value that it may return. The details of the wait-free quiescent HI SWSR multi-valued register from binary registers appear in [7].

In the full version of this paper [7] we also prove that it is essential for the reader to write to shared memory, otherwise it is impossible to obtain even a quiescent HI wait-free implementation. [Only the proposition itself is in the body of the paper, no proof. I suggest just removing the proposition and saying that in the full version we prove ….]

5 HI IMPLEMENTATIONS FROM SMALLER BASE OBJECTS

In this section, we show that for a large class of objects, a reasonably strong notion of history independence— *state-quiescent history independence* (Definition 5)—cannot be achieved from smaller base objects, if we require wait-freedom.

5.1 The class C_t

Informally, our impossibility result applies to all objects with the following properties:

- The object has a "non-trivial" read operation, which is able to distinguish between t different subsets of the object's possible states; and

[2] This is true in the sense that a READ operation that does not overlap with a WRITE will return the smallest index $i \in [K]$ such that $A[i] = 1$; for a READ that does overlap with a WRITE, the picture is much more complicated [45].

- The object can be "moved freely" from any state to any other state, in a single operation.

In fact, the impossibility result applies to other objects, including a queue, which do not fall into this class because they cannot be moved from any possible state to any other possible state in a single operation; for example, if a queue currently has two elements, we cannot reach the state where it is empty in one operation. For simplicity, we present here the proof for the simpler, more restricted class described above, and we discuss a generalization for a queue in the full version of this paper [7].

The class C_t is formally defined as follows:

Definition 7 (The class C_t). *An object O is in the class C_t for $t \geq 2$ if its state space Q can be partitioned into t nonempty subsets $X_1, \ldots, X_t$, such that*

- *The object has some operation o_{read}, which does not change the state of the object, such that for any two states $q_i \in X_i, q_j \in X_j$ where $i \neq j$, the response to o_{read} from state q_i differs from the response to o_{read} from state q_j.*
- *For any two states $q \neq q' \in Q$ there is some operation $o_{chg}(q, q')$ that causes the object to transition from state q to state q'.*

An object in the class C_t has at least t different states, and any pair of states are mutually reachable from each other by a single operation. Thus, the memory representations that arise from an implementation of an object in C_t, $t \geq 3$, from base objects with fewer than t states cannot all be at distance 1 from each other. By Proposition 4 we obtain:

Proposition 8. *There is no obstruction-free perfect HI implementation of an object in C_t, $t \geq 3$, from base objects with fewer than t states.*

Examples of objects in the class C_t. A t-valued read/write register is in the class C_t: it has t different states, each representing the value the register holds, and the Read operation distinguishes between them; the Write operation can move us from any state (i.e., any value) to any other state in a single operation. We have already seen in Section 4 that multi-valued registers *can* be implemented from binary registers, if we weaken either the progress or the history independence requirements. Our results for SWSR multi-valued registers are summarized in Table 1.

Another example of an object in the class C_t is a t-valued CAS object that supports a read operation: the state is again the current value of the CAS, and the read operation distinguishes between all t possible values; as for the o_{chg} operation, we can move from any state q to any state q' by invoking $\text{CAS}(X, q, q')$.

To illustrate the importance of the state-connectivity requirement in the definition of the class C_t, we argue that a *max register* [6], returning the maximum value ever written in it, is *not* in the class. The state space of a max register is not well-connected: as soon as we visit state m, the object can never go back to a state smaller than m. A simple modification to Vidyasankar's algorithm, where the writer only writes to A if the new value is bigger than all the values it has written in the past, results in a wait-free state-quiescent HI max register from binary registers.

Another object that is not in the class C_t is a *set* over t elements, with insert, remove and lookup operations. Even though the set has 2^t possible abstract states, its operations return only two responses, "success" or "failure"; thus, we cannot *distinguish* via a single operation between all 2^t states, or even between t states (the number of elements that could be in the set). There is a simple wait-free perfect HI implementation of a set over the domain $\{1, \ldots, t\}$, using t binary registers: we simply represent the set as an array S of length t, with $S[i] = 1$ if and only if element i is in the set, with the obvious implementation of insert, delete and lookup.

5.2 Impossibility of Wait-Free, State-Quiescent HI Implementations for the Class C_t

Let $O \in C_t$ be a high-level object with state space Q. Consider a wait-free state-quiescent HI implementation of O using $m \geq 1$ base objects $obj_1, \ldots, obj_m$. For each base object obj_i, let Q_i be the state space of obj_i; we assume that $|Q_i| \leq t - 1$. This is the only assumption we make about the base objects, and our impossibility result applies to arbitrary read-modify-write[3] base objects as well as to simple read-write base registers. Let h be a linearization function for the implementation.

We consider executions with two processes: One is a "reader" process r, which executes a single o_{read} operation, and the other is a "changer" process c, which repeatedly invokes o_{chg} operations.

For the purpose of the impossibility result, we assume, that the local state of a process p_i contains the complete history of p_i's invocations and responses. Our goal is to show that we can construct an execution where r does not return from its single o_{read} operation, violating wait-freedom.

The executions that we construct have the following form:

$$\alpha_{q_0,\ldots,q_k} = o_{chg}(q_0, q_1), r_1, o_{chg}(q_1, q_2), r_2, \ldots, o_{chg}(q_{k-1}, q_k), r_k$$

where $o_{chg}(q_i, q_{i+1})$ is an operation executed by the changer process during which the reader process takes no steps, and r_i is a single step by the reader process. The reader executes a single o_{read} operation that is invoked immediately after the first o_{chg} operation completes, and we will argue that the reader never returns.

In any linearization of $\alpha_{q_0,\ldots,q_k}$, the operations $o_{chg}(q_0, q_1), \ldots, o_{chg}(q_{k-1}, q_k)$ must be linearized in order, as they do not overlap. Furthermore, the o_{read} operation carried out by the reader is not state-changing. Thus, the linearization of $\alpha_{q_0,\ldots,q_k}$ ends with the object in state q_k, and we abuse the terminology by saying that the execution "ends at state q_k".

We say that execution $\alpha_{q_0,\ldots,q_k}$ *avoids* a subset $X \subseteq Q$ if $\{q_1, \ldots, q_k\} \cap X = \emptyset$. (Note that we may have $q_0 \in X$ and still say that $\alpha_{q_0,\ldots,q_k}$ avoids X; this is fine for our purposes, because the reader only starts running after the first o_{chg} operation completes.)

Lemma 9. *There exists a partition of the possible return values R for the o_{read} into subsets $R_1, \ldots, R_t$,[4] such that if an execution $\alpha_{q_0,\ldots,q_k}$ avoids $X_i \subseteq Q$, then the o_{read} operation cannot return any value from R_i at any point in $\alpha_{q_0,\ldots,q_k}$.*

Proof. For each $1 \leq i \leq t$, let R_i be the set of values ρ such that for some state $q \in X_i$, the o_{read} operation returns ρ when executed from state q. By the definition of the class C_t, the sets $R_1, \ldots, R_t$

[3] Some examples of read-modify-write objects are CAS, test-and-set and swap objects.
[4] We assume there are no unused values in R, that is, for any $r \in R$, there is some state $q \in Q$ such that when o_{read} is executed from state q, it returns r.

Perfect HI (Def. 3)	State-quiescent HI (Def. 5)	Quiescent HI (Def. 6)	Progress
Impossible (Prop. 8)	Impossible (Cor. ??)	Possible (see [7])	Wait-free
Impossible (Prop. 8)	Possible (see [7])	Possible (see [7])	Lock-free

Table 1: Summary of results for implementing a SWSR multi-valued register from binary registers

are disjoint, and since the sets $X_1, \ldots, X_t$ partition the state space Q, the sets $R_1, \ldots, R_t$ partition the set of responses R.

Fix an execution $\alpha_{q_0,\ldots,q_k}$ that avoids X_i, and recall that in any linearization, the operations $o_{chg}(q_0, q_1), \ldots, o_{chg}(q_{k-1}, q_k)$ must be linearized in-order, as they are non-overlapping operations by the same process. The operation o_{read} cannot be linearized before the first operation $o_{chg}(q_0, q_1)$, because it is invoked after this operation completes. Thus, o_{read} either does not return in $\alpha_{q_0,\ldots,q_k}$, or it is linearized after some operation $o_{chg}(q_j, q_{j+1})$ where $j \geq 0$. In the latter case, let $\ell \neq i$ be the index such that $q_{j+1} \in X_\ell$; we know that $\ell \neq i$ as $\alpha_{q_0,\ldots,q_k}$ avoids X_i. The value returned by the o_{read} is in the set R_ℓ, which is disjoint from R_i. Therefore in $\alpha_{q_0,\ldots,q_k}$ the o_{read} operation either does not return, or returns a value that is not in R_i. □

Using the fact that each base object has at most $t-1$ possible states, we can construct t arbitrarily long executions that the reader cannot distinguish from one another, such that each subset X_i is avoided by one of the t executions. Two execution prefixes α_1 and α_2 are *indistinguishable* to the reader, denoted $\alpha_1 \overset{r}{\sim} \alpha_2$, if the reader is in the same state in the final configurations of α_1 and α_2.

The construction is inductive, with each step extending the executions by one operation and a single step of the reader:

Lemma 10. *Fix $k \geq 0$, and suppose we are given t executions of the form $\alpha_i = \alpha_{q_0^i,\ldots,q_k^i}$ for $i = 1, \ldots, t$, such that $\alpha_1 \overset{r}{\sim} \ldots \overset{r}{\sim} \alpha_t$, and each α_i avoids X_i. Then we can extend each α_i into an execution $\alpha'_i = \alpha_{q_0^i,\ldots,q_{k+1}^i}$ that also avoids X_i, such that $\alpha'_1 \overset{r}{\sim} \ldots \overset{r}{\sim} \alpha'_t$.*

Proof. Let $\alpha_i = \alpha_{q_0^i,\ldots,q_k^i}$ for $i = 1, \ldots, t$ be executions satisfying the conditions of the lemma, and let us construct extensions $\alpha'_i = \alpha_{q_0^i,\ldots,q_k^i,q_{k+1}^i}$ for each $i = 1, \ldots, t$. By assumption, the reader is in the same local state at the end of all executions α_i for $1 \leq i \leq t$, and so its next step is the same in all of them. Our goal is to choose a next state q_{k+1}^i for each $i = 1, \ldots, t$, and extend each $\alpha_i = \alpha_{q_0^i,\ldots,q_k^i}$ into $\alpha'_i = \alpha_{q_0^i,\ldots,q_k^i,q_{k+1}^i}$ by appending an operation $o_{chg}(q_k^i, q_{k+1}^i)$, followed by a single step of the reader. We must do so in a way that continues to avoid X_i, and maintains indistinguishability to the reader.

Since the implementation of O is state-quiescent HI and since each execution α_i ends in a state-quiescent configuration, if α_i ends in state q, then the memory must be in its canonical representation, $can(q)$ (as defined in Section 2).

Let obj_ℓ be the base object accessed by the reader in its next step in all t executions. Because obj_ℓ has only $t-1$ possible memory states and there are t subsets $X_1, \ldots, X_t$, there must exist two distinct subsets $X_j, X_{j'}$ $(j \neq j')$ and two states $q \in X_j, q' \in X_{j'}$ such that $can(q)[\ell] = can(q')[\ell]$. For every $1 \leq i \leq t$, there is a state $q_{k+1}^i \in \{q, q'\}$ such that $q_{k+1}^i \notin X_i$: if $i \notin \{j, j'\}$ then we choose between q and q' arbitrarily, and if $i = j$ or $i = j'$ then we choose $q_{k+1}^i = q'$ or $q_{k+1}^i = q$, respectively.

We extend each $\alpha_i = \alpha_{q_0^i,\ldots,q_k^i}$ into $\alpha'_i = \alpha_{q_0^i,\ldots,q_k^i,q_{k+1}^i}$ by appending a complete $o_{chg}(q_k^i, q_{k+1}^i)$ operation, followed by a single step of the reader. The resulting execution α'_i still avoids X_i, as we had $\left\{q_1^i, \ldots, q_k^i\right\} \cap X_i = \emptyset$, and the new state also satisfies $q_{k+1}^i \notin X_i$. Moreover, when the reader takes its step, it observes the same state for the base object obj_ℓ that it accesses in all executions, as all of them end in either state q or state q', and $can(q)[\ell] = can(q')[\ell]$. Therefore, the reader cannot distinguish the new executions from one another. □

By repeatedly applying Lemma 10, we can construct arbitrarily long executions, with the reader taking more and more steps (since in an execution $\alpha_{q_0,\ldots,q_k}$ the reader takes k steps) but never returning. This is because for each return value there is an execution avoiding it, and by Lemma 9 and since all the executions are indistinguishable to the reader, no value can be returned. This contradicts the wait-freedom of the implementation, to yield (the proof appears in [7]):

Theorem 11. *For any object O in the class C_t, $t \geq 2$, there is no wait-free implementation that is state-quiescent HI using base objects with fewer than t states.*

6 A HI UNIVERSAL IMPLEMENTATION

In the previous sections, we considered history-independent implementations of objects from base objects that are too small to store the state of the abstract object in its entirety, and showed that certain tradeoffs are unavoidable in this setting: for many objects, one must sacrifice either wait-freedom or state-quiescent history independence. We now turn to study large base objects, which can store the entire state of the abstract object, together with auxiliary information; we show that in this regime, a wait-free implementation that is state-quiescent HI *is* possible. Our implementation actually satisfies a somewhat stronger property than state-quiescent HI: at any point in the execution, the observer cannot gain information about operations that completed before a pending operation started, except for the state of the object when the earliest pending operation began.

When the full state of the object can be stored in a single memory cell, there is a simple lock-free universal implementation, using *load-link/store-conditional* (LLSC).[5] The current state of the object is stored in a single cell, an operation reads the current value of this cell, using *LL*, and then tries to write the new value of the object (after applying its changes) in this cell, using *SC*.

[5] In hardware, *load-linked* reads a memory cell, while *store-conditional* changes this memory cell, provided that it was not written since the process' most recent load-linked.

This implementation is clearly perfect HI. However, it is not wait-free since an operation may repeatedly fail, since other operations may modify the memory cell in between its LL and its SC. The standard way to make the universal implementation wait-free relies on *helping* [19, 27]: When starting, an operation *announces* its type and arguments in shared memory. Operations check whether other processes have pending operations and help them to complete, obtaining the necessary information from their announcement; after helping an operation to complete, they store a response to be returned later. This breaches history independence, revealing the type and arguments of prior and pending operations, as well as the responses of some completed operations.

Our wait-free, history-independent universal implementation follows a similar approach, but ensures that announcements and responses are cleared before operations complete, to guarantee that forbidden information are not left in shared memory. Care is take to erase information only after it is no longer needed.

To use the more standard and commonly-available atomic CAS, we implement an abstraction of a *context-aware* variant of LLSC [31], which explicitly manages the set of processes that have load-linked this cell as *context*. This again breaches history independence, as the context reveals information about prior accesses. To erase this information, the implementation clears the context of a memory cell using an additional *release* operation, added to the interface of context-aware *releasable LLSC* (R-LLSC). This yields a *wait-free state-quiescent HI universal implementation from atomic CAS*.

The next section presents the universal implementation using linearizable R-LLSC objects. We then give a lock-free R-LLSC perfect HI implementation from atomic CAS (Section 6.2), and obtain a *wait-free state-quiescent HI universal* implementation from atomic CAS, in Section 6.3.

6.1 Universal HI Implementation from Linearizable Releasable LLSC

A context-aware *load-link/store-conditional* (LLSC) object over a domain V is defined as follows: the state of an LLSC object O is the pair $(O.val, O.context)$, where $O.val \in V$ is the value of the object, and $O.context$ is a set of processes. The initial state is $(v_0, \emptyset)$, where $v_0 \in V$ is a designated initial value. Process p_i can perform the following operations:

LL(O)**:** adds p_i to $O.context$ and returns $O.val$.

VL(O)**:** returns *true* if $p_i \in O.context$ and *false* otherwise.

SC(O, new)**:** if $p_i \in O.context$, sets $O.val = new$ and $O.context = \emptyset$, and returns *true*; otherwise, it returns *false*.

Load(O)**:** returns $O.val$ without changing $O.context$.

Store(O, new)**:** sets $O.val = new$ and $O.context = \emptyset$ and returns *true*.

A VL, SC or Store is *successful* if it returns *true*; note that Store is always successful. Load and Store operations are added to simplify the code and proof.

The universal implementation appears in Algorithm 1. The right side of Lines 6, 18 and 25, marked in blue, as well as Lines 22 and 27, marked in red, are only used to ensure history independence; we ignore them for now, and explain their usage later. We assume that the set of possible responses R of the object is disjoint from its set of operations O, i.e., $R \cap O = \emptyset$, and $\perp \notin R \cup O$.

Algorithm 1 State-quiescent HI universal implementation from R-LLSC: code for process p_i

$head$, R-LLSC variable initialized to $\langle q_0, \perp \rangle$, where q_0 is the initial state
$announce[n]$, R-LLSC variable array all cells initialized to $\perp$
local $priority_i$, initialized to i

ApplyReadOnly$(op \in O)$: ▷ Read-only operations
1: $\langle q, \_ \rangle \leftarrow$ Load$(head)$
2: $\_, rsp \leftarrow \Delta(q, op)$
3: **return** rsp

Apply$(op \in O)$: ▷ State-changing operations
4: Store$(announce[i], op)$
5: **while** Load$(announce[i]) \notin R$ **do**
6L: $\langle q, r \rangle \leftarrow$ LL$(head)$ ‖ 6R.1: **wait until** Load$(announce[i]) \notin R$
6R.2: goto Line 24
7: **if** $r = \perp$ **then** ▷ In-between operations
8: $help \leftarrow$ Load$(announce[priority_i])$
9: **if** $help \in O$ **then** $apply\text{-}op \leftarrow help$; $j \leftarrow priority_i$
↪ ▷ Try to apply another process operation
10: **else**
11: **if** Load$(announce[i]) \notin O$ **then** continue
↪ ▷ Go to the beginning of the loop
12: $apply\text{-}op \leftarrow op$; $j \leftarrow i$
↪ ▷ Try to apply your own operation
13: $state, rsp \leftarrow \Delta(q, apply\text{-}op)$
14: **if** SC$(head, \langle state, \langle rsp, j \rangle \rangle)$ **then** ▷ End of the first stage
15: $priority_i \leftarrow (priority_i + 1) \bmod n$
16: **else**
17: $\langle rsp, j \rangle \leftarrow r$
18L]: $a \leftarrow$ LL$(announce[j])$
↪ ‖ 18R.1: **wait until** Load$(announce[i]) \notin R$
18R.2: RL$(announce[j])$
18R.3: goto Line 24
19: **if** VL$(head) = true$ **then**
20: **if** $a \in O$ **then** SC$(announce[j], rsp)$
↪ ▷ End of the second stage
21: SC$(head, \langle q, \perp \rangle)$ ▷ End of the third stage
22: **if** $a = \perp$ **then** RL$(announce[j])$
23: continue ▷ Go to the beginning of the loop
24: $response \leftarrow$ Load$(announce[i])$
25L: $\langle q, r \rangle \leftarrow$ LL$(head)$ ‖ 25R.1: **wait until** Load$(head) \neq \langle _, \langle _, i \rangle \rangle$
25R.2: goto Line 27
26: **if** $r = \langle \_, i \rangle$ **then** SC$(head, \langle q, \perp \rangle)$
↪ ▷ Clear response from head before returning
27: **else** RL$(head)$
28: Store$(announce[i], \perp)$ ▷ Clear response from $announce[i]$
29: **return** $response$

An array $announce[1..n]$ stores information about pending operations, while $head$ holds the current state of the object, along with some auxiliary information, like the response to the most recently applied operation o, and the identifier of the process that invoked o. In-between operations, the value of $head$ is $\langle q, \perp \rangle$, where q is the current state of the object.

A process invoking a read-only operation calls ApplyReadOnly, which simply reads the object's state from *head* and returns a response according to the sequential specification of the object. This does not change the memory representation of the implementation.

A process p_i invoking a state-changing operation calls Apply. First, p_i announces the operation by writing its description to $announce[i]$, and then, p_i repeatedly tries to apply operations (either its own operation or operations announced by other processes) until it identifies that its own operation has been applied. The choice of which operation to apply (Lines 8-12) is determined by a local variable $priority_i$, which is not part of the memory representation. If there is a pending operation by process p_j, $j = priority_i$, then p_i applies p_j's operation; otherwise, it applies its own operation. Each time p_i successfully changes the state of the object, it increments $priority_i$ (modulo n). This ensures that all pending operations will eventually help the same process.

Applying an operation o, with $\Delta(q, o) = (q', r)$, consists of three stages, each of which can be performed by any process (not just the process that invoked o, and not necessarily the same process for all three stages).

In the first stage, head is changed from $\langle q, \perp\rangle$ to $\langle q', \langle r, j\rangle\rangle$, where p_j is the process that invoked operation o. The stage starts when some process p_i reads $\langle q, \perp\rangle$ from *head* with LL(*head*) (Line 6L), and decides which operation to try to apply, say o by process p_j. To do so, p_i performs SC($head, \langle q', \langle r, j\rangle\rangle$) (Line 14). If the SC is successful, the value of *head* did not change between the LL and the SC of p_i. This ensures that the chosen operation, read from $announce[j]$ in Line 8 or Line 11 after the LL(*head*), is not applied more than once.

In the second stage, the response r is written into $announce[j]$, overwriting o itself, to notify the invoking process p_j that its operation was performed, and what value p_j should return. A process p_i that writes the response to $announce[j]$ has read $\langle q', \langle r, j\rangle\rangle$ from *head* with LL(*head*) (Line 6L) and then performs a successful VL(*head*) in Line 19. If p_i performs a successful SC($announce[j], r$) in Line 20, then it previously performed a LL($announce[j]$) in Line 18L, i.e., between LL(*head*) and VL(*head*). This guarantees that the value of *head* does not change between LL(*head*) and SC($announce[j], r$).

The third and final stage changes *head* from $\langle q', \langle r, j\rangle\rangle$ to $\langle q', \perp\rangle$. This erases the response r and the process index j, ensuring that forbidden information about the history is not revealed. The invoking process p_j does not return until its response is cleared from *head* (Lines 25L and 26). This ensures that a successful SC($announce[j], r$) (Line 20) writes the right response to the applied operation, since it can only occur before the response value is cleared from *head*. If a process performs a successful SC($head, \langle q', \perp\rangle$) (Line 21), then the previous LL(*head*) (Line 6L) guarantees that the replaced value of *head* was indeed of the form $\langle q', r'\rangle$, where $r' \neq \perp$.

Finally, before returning, p_j also clears $announce[j]$ (Line 28).

Achieving history independence. Algorithm 1, without the lines shown in red, *is not state-quiescent HI*, and in fact it is not even quiescent HI: although we delete past responses from the *head* and clear $announce[i]$ before returning, their *context* fields may reveal information about the history even when no operation is pending. For example, suppose process p_i invokes an operation o and writes it to $announce[i]$, and begins the main loop where it tries to perform operations. Before p_i can even perform LL(*head*) in Line 6L, faster processes carry out operation o and all other pending operations, and return. When p_i does reach Line 6L and calls LL(*head*), it sees that the system is in-between operations ($head = \langle q, \perp\rangle$), and it finds no other processes requiring help. It thus returns straightaway, leaving its link in the *context* field of *head*. This might seem innocuous, but it could, for example, reveal that a counter supporting fetch-and-increment and fetch-and-decrement operations, whose current value is zero, was non-zero in the past, because the observer can see that *some* state-changing operation was performed on it.

To address this problem, we add a release (RL) operation to the LLSC object. RL removes a process from the context, and we use it to ensure that the *context* component of each LLSC object in the implementation is empty in a state-quiescent configuration. Formally, a *releasable LLSC* (R-LLSC) adds the following operation, performed by process p_i:

RL(O): removes p_i from $O.context$ and returns *true*.

RL operations are added in Lines 22 and 27 of Algorithm 1, both marked in red. We show below a lock-free implementation of an R-LLSC object from atomic CAS. The implementation is not wait-free, as RL operations may interfere with other ongoing operations (including LL). To handle R-LLSC operations that may block and obtain a wait-free universal HI implementation, we add the code marked in blue, in Lines 6, 18 and 25. These lines interleave steps in which process p_i checks whether some other process p_j has already accomplished what p_i was trying to do (e.g., p_j applied p_i's operation for it). The notation ‖ indicates the interleaving of steps between the code appearing to its left and to its right, with some unspecified but finite number of steps taken on each side before the process switches and starts taking steps of the other side.

In Line 18R.2, a RL ensures that if p_i's operation is performed by another process while p_i itself is trying to help a third process p_j, then the LL($announce[j]$) in Line 18L leaves no trace. We must do this because we do not know whether the LL($announce[j]$) on the left side has already "taken effect" or not at the point where the **wait until** command on the right-hand side is done.

The proof partitions the execution into segments, with each successful state-change (that is, each successful SC($head, \langle q, \langle r, i\rangle\rangle$)) beginning a new segment. We linearize exactly one state-changing operation at the beginning of each such segment, and interleave the linearization points of the read-only operations according to the segment in which they read the *head*. The full linerizability proof, along with additional properties of Algorithm 1, appears in [7].

Theorem 12. *Algorithm 1 is a linearizable universal implementation from linearizable R-LLSC objects.*

6.2 Lock-Free Perfect-HI R-LLSC Object from Atomic CAS

The implementation of an R-LLSC object using a single atomic CAS object is based on [30], and its code appears in Algorithm 2. The state of the R-LLSC object O is stored in the CAS object in the format $x = (v, c_1, \ldots, c_n) \in V \times \{0, 1\}^n$, where $v = O.val$ is its value, and each bit c_i indicates whether or not $p_i \in O.context$. Denote $x.val = v$ and $x.context[i] = c_i$. The implementation is perfect HI,

Algorithm 2 Lock-free perfect HI R-LLSC from CAS: code for process p_i

X: CAS variable initialized to $(v_0, 0, \ldots, 0)$

```
    LL(O):
1:  cur ← Read(X)
2:  new ← cur
3:  new.context[i] ← 1
4:  while !CAS(X, cur, new) do
5:      cur ← Read(X)
6:      new ← cur
7:      new.context[i] ← 1
8:  return cur.val

    SC(O, v):
9:  cur ← Read(X)
10: while cur.context[i] = 1 do
11:     if
    ↪ CAS(X, cur, (v, 0, ..., 0))
    ↪ then return true
12:     cur ← Read(X)
13: return false

    VL(O):
14: cur ← Read(X)
15: return cur.context[i]

    RL(O):
16: cur ← Read(X)
17: new ← cur
18: new.context[i] ← 0
19: while cur.context[i] = 1 do
20:     if CAS(X, cur, new) then
21:         return true
22:     cur ← Read(X)
23:     new ← cur
24:     new.context[i] ← 0
25: return true

    Load(O):
26: cur ← Read(X)
27: return cur.val

    Store(O, v):
28: Write(X, (v, 0, ..., 0))
29: return true
```

because the mapping from abstract state to memory representation is unique, and no additional information is stored.

The operations Load and VL are read-only; to implement them, we simply read X and return the appropriate response. A Store operation writes into the CAS a new value with an empty context, regardless of the current state of the objects. Finally, the LL, RL and SC operations are implemented by reading X and then trying to update it using a CAS operation, but this is not guaranteed to succeed; hence, these operations are only lock-free, not wait-free.

The proof of the next theorem appears in [7].

Theorem 13. *Algorithm 2 is a lock-free linearizable perfect HI implementation of a R-LLSC object from atomic CAS.*

When considering progress, we cannot rely on the progress of R-LLSC algorithm as a black box, because the LL, RL and SC operations are not by themselves wait-free. Still, we can rely on the *interactions* among the R-LLSC operations to ensure that the way they are used in Algorithm 1 is wait-free. The SC and Store operations "help" RL and SC operations, in the sense that a successful SC or Store operation clears the *context*, causing all pending RL and SC operations to complete: RL operations return because the process is indeed no longer in the *context*, and SC operations return because they have failed. We have the next lemma, proved in [7]:

Lemma 14. *Let op be an RL or SC operation that is pending in execution α, and suppose that in α, a SC or Store operation is invoked after op, and returns true before op returns. Then in any extension of α, op returns within a finite number of steps by the process that invoked it.*

6.3 Wait-Free State-Quiescent HI Universal Implementation from Atomic CAS

We now combine Algorithm 1 with an R-LLSC implementation of Section 6.2, to get a wait-free state-quiescent HI universal implementation, despite the R-LLSC implementation being only lock-free.

Since LL, RL and SC are lock-free, if a process tries to modify *head*, some process will eventually succeed in modifying *head*. By Lemma 14, this allows other pending SC and RL operations to complete. This property does not hold for a LL operation, which may never return, but this is handled by the invoking algorithm, which interleaves steps that check if the operation was performed by a different process, as explained above. The helping mechanism ensures that every pending operation is eventually applied. Thus, the wait conditions in Lines 6L, 18L and 25L are eventually false, releasing operations that might be stuck in an LL operation.

Finally, we discuss history independence. At a state-quiescent configuration, the states of the R-LLSC objects are uniquely defined, according to the state reached by the sequence of operations applied during the execution. Note that the state includes both the *val* and *context* part of the object. Since the R-LLSC implementation is perfect HI, by a simple composition, this state translates to a unique memory representation. The implementation of the R-LLSC objects provides the strongest form of history independence, and for our need, a weaker state-quiescent HI implementation also suffices. The next theorem, proved in [7], concludes this section by putting all the pieces together.

Theorem 15. *Any abstract object has a wait-free state-quiescent HI implementation from CAS base objects that can store $O(s + 2^n)$ values.*

7 DISCUSSION

This paper introduces the notion of history independence for concurrent data structures, explores various ways to define it, and derives possibility and impossibility results. We gave two main algorithmic results: a wait-free multi-valued register, and a universal implementation of arbitrary objects. Interestingly, both implementations follow a similar recipe: starting with a history-independent lock-free implementation, helping is introduced to achieve wait-freedom. However, helping tends to leak information about the history of the object, so we introduce mechanisms to clear it.

Our results open up a range of research avenues, exploring history-independent object implementations and other notions of history independence. *Randomization* is of particular importance, as it is a tool frequently used to achieve both algorithmic efficiency and history independence. When randomization is introduced, the distinction between weak and strong history independence becomes meaningful. We note that randomization will not help circumvent the impossibility result from Section 5.2, if we require *strong* history independence: by a result of [24, 25], in any strongly history-independent implementation of a *reversible object*[6], the canonical memory representation needs to be fixed up-front, and our impossibility proof then goes through. However, this does not rule out *weakly* history-independent implementations. We remark that

[6]For example, a register is reversible, while an increment-only counter is not.

even coming up with a meaningful definition for history independence in randomized concurrent implementations is non-trivial, because randomization can affect the number of steps an operation takes, making it challenging to define a probability distribution over the memory states at the points where the observer is allowed to observe the memory.

ACKNOWLEDGMENTS

Hagit Attiya is partially supported by the Israel Science Foundation (grant number 22/1425). Rotem Oshman is funded by NSF-BSF Grant No. 2022699. Michael Bender and Martín Farach-Colton are funded by NSF Grants CCF-2106999, CCF-2118620, CNS-1938180, CCF-2118832, CCF-2106827, CNS-1938709, CCF-2247577.

REFERENCES

[1] Umut A Acar, Guy E Blelloch, Robert Harper, Jorge L Vittes, and Shan Leung Maverick Woo. 2004. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 531–540.

[2] Ole Amble and Donald Ervin Knuth. 1974. Ordered hash tables. *Comput. J.* 17, 2 (1974), 135–142.

[3] Arne Andersson and Thomas Ottmann. 1991. Faster uniquely represented dictionaries. In *Proc. of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 642–649.

[4] Arne Andersson and Thomas Ottmann. 1995. New tight bounds on uniquely represented dictionaries. *SIAM J. Comput.* 24, 5 (1995), 1091–1103.

[5] Cecilia R Aragon and Raimund G Seidel. 1989. Randomized search trees. In *Proc. of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 540–545.

[6] James Aspnes, Hagit Attiya, and Keren Censor. 2009. Max registers, counters, and monotone circuits. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*. 36–45.

[7] Hagit Attiya, Michael A. Bender, Martin Farach-Colton, Rotem Oshman, and Noa Schiller. 2024. *History-Independent Concurrent Objects*. Technical Report 2403.14445. arXiv. https://doi.org/10.48550/arXiv.2403.14445 Full version of this paper.

[8] Sumeet Bajaj, Anrin Chakraborti, and Radu Sion. 2015. The Foundations of History Independence. *arXiv preprint arXiv:1501.06508* (2015).

[9] Sumeet Bajaj, Anrin Chakraborti, and Radu Sion. 2016. Practical Foundations of History Independence. *IEEE Trans. Inf. Forensics Secur.* 11, 2 (2016), 303–312. https://doi.org/10.1109/TIFS.2015.2491309

[10] Sumit Bajaj and Radu Sion. 2013. Ficklebase: Looking into the future to erase the past. In *Proc. of the 29th IEEE International Conference on Data Engineering (ICDE)*. 86–97.

[11] Sumeet Bajaj and Radu Sion. 2013. HIFS: History independence for file systems. In *Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*. 1285–1296.

[12] Michael A. Bender, Jon Berry, Rob Johnson, Thomas M. Kroeger, Samuel McCauley, Cynthia A. Phillips, Bertrand Simon, Shikha Singh, and David Zage. 2016. Anti-Persistence on Persistent Storage: History-Independent Sparse Tables and Dictionaries. In *Proc. 35th ACM Symposium on Principles of Database Systems (PODS)*. 289–302.

[13] Michael A. Bender, Alex Conway, Martín Farach-Colton, Hanna Komlós, William Kuszmaul, and Nicole Wein. 2022. Online List Labeling: Breaking the $\log^2 n$ Barrier. In *Proc. 63rd IEEE Annual Symposium on Foundations of Computer Science (FOCS)*.

[14] John Bethencourt, Dan Boneh, and Brent Waters. 2007. Cryptographic methods for storing ballots on a voting machine. In *Proc. of the 14th Network and Distributed System Security Symposium (NDSS)*.

[15] Guy E Blelloch and Daniel Golovin. 2007. Strongly history-independent hashing with applications. In *Proc. of the 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 272–282.

[16] Niv Buchbinder and Erez Petrank. 2003. Lower and Upper Bounds on Obtaining History Independence. In *Advances in Cryptology*. 445–462.

[17] Pedro Celis, Per-Åke Larson, and J. Ian Munro. 1985. Robin Hood Hashing (Preliminary Report). In *26th Annual Symposium on Foundations of Computer Science (FOCS'85)*. Portland, Oregon, USA, 281–288. https://doi.org/10.1109/SFCS.1985.48

[18] Bo Chen and Radu Sion. 2015. HiFlash: A History Independent Flash Device. *CoRR* abs/1511.05180 (2015). arXiv:1511.05180 http://arxiv.org/abs/1511.05180

[19] Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A highly-efficient wait-free universal construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 325–334. https://doi.org/10.1145/1989493.1989549

[20] Daniel Golovin. 2008. *Uniquely Represented Data Structures with Applications to Privacy*. Ph. D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, 2008.

[21] Daniel Golovin. 2009. B-treaps: A uniquely represented alternative to B-Trees. In *Proc. of the 36th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*. Springer Berlin Heidelberg, 487–499.

[22] Daniel Golovin. 2010. The B-skip-list: A simpler uniquely represented alternative to B-trees. *arXiv preprint arXiv:1005.0662* (2010).

[23] Michael T. Goodrich, Evgenios M. Kornaropoulos, Michael Mitzenmacher, and Roberto Tamassia. 2017. Auditable Data Structures. In *Proc. IEEE European Symposium on Security and Privacy (EuroS&P)*. 285–300. https://doi.org/10.1109/EuroSP.2017.46

[24] Jason D. Hartline, Edwin S. Hong, Alexander E. Mohr, William R. Pentney, and Emily Rocke. 2002. Characterizing History Independent Data Structures. In *Proceedings of the Algorithms and Computation, 13th International Symposium (ISAAC)*. 229–240. https://doi.org/10.1007/3-540-36136-7_21

[25] Jason D Hartline, Edwin S Hong, Alexander E Mohr, William R Pentney, and Emily C Rocke. 2005. Characterizing history independent data structures. *Algorithmica* 42, 1 (2005), 57–74.

[26] M. Herlihy. 1990. A Methodology for Implementing Highly Concurrent Data Structures. *SIGPLAN Not.* 25, 3 (feb 1990), 197–206. https://doi.org/10.1145/99164.99185

[27] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems* 13, 1 (jan 1991), 124–149. https://doi.org/10.1145/114005.102808

[28] Maurice Herlihy. 1993. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems* 15, 5 (nov 1993), 745–770. https://doi.org/10.1145/161468.161469

[29] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972

[30] Amos Israeli and Lihu Rappoport. 1994. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing* (Los Angeles, California, USA) *(PODC '94)*. Association for Computing Machinery, New York, NY, USA, 151–160. https://doi.org/10.1145/197917.198079

[31] Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. 2023. Durable Algorithms for Writable LL/SC and CAS with Dynamic Joining. In *37th International Symposium on Distributed Computing (DISC 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 281)*, Rotem Oshman (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:20. https://doi.org/10.4230/LIPIcs.DISC.2023.25

[32] Daniele Micciancio. 1997. Oblivious data structures: applications to cryptography. In *Proc. of the 29th Annual ACM Symposium on Theory of Computing (STOC)*. 456–464.

[33] Tal Moran, Moni Naor, and Gil Segev. 2007. Deterministic history-independent strategies for storing information on write-once memories. In *Proc. 34th International Colloquium on Automata, Languages and Programming (ICALP)*.

[34] Moni Naor, Gil Segev, and Udi Wieder. 2008. History-independent cuckoo hashing. In *Proc. of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*. Springer, 631–642.

[35] Moni Naor and Vanessa Teague. 2001. Anti-persistence: history independent data structures. In *Proc. of the 33rd Annual ACM Symposium on Theory of Computing (STOC)*. 492–501.

[36] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2016. Arx: A Strongly Encrypted Database System. *IACR Cryptol. ePrint Arch.* (2016), 591. http://eprint.iacr.org/2016/591

[37] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[38] William Pugh and Tim Teitelbaum. 1989. Incremental computation via function caching. In *Proc. of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 315–328.

[39] Daniel S Roche, Adam J Aviv, and Seung Geol Choi. 2015. Oblivious Secure Deletion with Bounded History Independence. *arXiv preprint arXiv:1505.07391* (2015).

[40] Daniel S. Roche, Adam J. Aviv, and Seung Geol Choi. 2016. A Practical Oblivious Map Data Structure with Secure Deletion and History Independence. In *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 178–197. https://doi.org/10.1109/SP.2016.19

[41] Julian Shun and Guy E. Blelloch. 2014. Phase-Concurrent Hash Tables for Determinism. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (Prague, Czech Republic) *(SPAA '14)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/2612669.2612687

[42] Lawrence Snyder. 1977. On uniquely represented data structures. In *Proc. of the 18th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 142–146.

[43] Rajamani Sundar and Robert Endre Tarjan. 1990. Unique binary search tree representations and equality-testing of sets and sequences. In *Proc. of the 22nd Annual ACM Symposium on Theory of Computing (STOC)*. 18–25.
[44] Theodoros Tzouramanis. 2012. History-independence: a fresh look at the case of R-trees. In *Proc. 27th Annual ACM Symposium on Applied Computing (SAC)*. 7–12.
[45] K. Vidyasankar. 1988. Converting Lamport's regular register to atomic register. *Inform. Process. Lett.* 28, 6 (1988), 287–290.