On Correctness of Single-Writer Multiple-Reader Data Structures [*]

Kfir Lev-Ari
EE Department
Technion, Haifa, Israel
kfirla@tx.technion.ac.il
+972-54-2664176

Gregory Chockler
CS Department
Royal Holloway, London, UK
gregory.chockler@rhul.ac.uk

Idit Keidar
EE Department
Technion, Haifa, Israel
idish@ee.technion.ac.il

Abstract

We study the correctness of shared data structures under read-write concurrency. A popular approach to ensuring correctness of a read-only operations in the presence of concurrent updates, is read-set validation, which checks that all read variables have not changed since they were first read. In practice, this approach is often too conservative, which adversely affects performance. In this paper, we introduce a new framework for reasoning about correctness of single-writer/multi-reader concurrent data structures, which replaces validation of the entire read-set with more general criteria. Namely, instead of verifying that all read shared variables still hold the values read from them, we verify abstract conditions over the shared variables, which we call base conditions. We show that obtaining a consistent snapshot of some base condition at every point in time implies correctness of read-only operations executing in parallel with updates. Somewhat surprisingly, the resulting correctness guarantees are weaker than linearizability, and instead captured through two new conditions: validity and regularity. Roughly, the former requires that a read-only operation never reaches a state unreachable in a sequential execution; and the latter is a generalization of the Lamport's notion of regularity for arbitrary data structures. We illustrate how our framework can be applied for reasoning about correctness of a variety of data structure implementations, such as linked lists as well as those implemented according to a popular read-copy-update (RCU) methodology. We further extend our framework to capture other commonly used correctness criteria, such as linearizability and sequential consistency.

Regular submission. If not selected, please consider for the brief announcement format.

Eligible to be considered for the best student paper award: Kfir Lev-Ari, the primary contributor, is a full time student.

[*]This work was partially supported by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), by the Israeli Ministry of Science, by a Royal Society International Exchanges Grant, and by the Randy L. and Melvin R. Berlin Fellowship in the Cyber Security Research Program.

1 Introduction

Motivation Concurrency is an essential part of computing nowadays. As part of the paradigm shift towards concurrency, we face a vast amount of legacy sequential code that needs to be parallelized. A key challenge for parallelization is verifying the correctness of the new or transformed code. There is a fundamental tradeoff between generality and performance in state-of-the-art approaches to correct parallelization. General purpose methodologies, such as transactional memory [14, 25] and coarse-grained locking, which do not take into account the inner workings of a specific data structure, are out-performed by hand-tailored fine-grained solutions [21]. Yet hand-tailored fine-grained solutions are notoriously difficult to develop and verify. In this work, we take a step towards mitigating this tradeoff.

It has been observed by many that correctly implementing concurrent modifications of a data structure is extremely hard, and moreover, contention among writers can severely hamper performance [23]. It is therefore not surprising that many approaches do not allow write-write concurrency; these include the *read-copy-update (RCU)* approach [20], flat-combining [13], coarse-grained readers-writer locking [9], and pessimistic software lock-elision [1]. It has been shown that such methodologies can perform better than ones that allow write-write concurrency, both when there are very few updates relative to queries [20] and when writes contend heavily [13]. We focus here on solutions that allow only read-read and read-write concurrency, which we call *single-writer/multiple-reader (SWMR)*.

A popular approach to ensuring correctness of read-only operations in the presence of concurrent update, is *read-set validitation*, which checks that no shared variables have changed since they were first read. In practice, this approach is often too conservative, which adversely affects performance. For example, when traversing a linked list, it suffices to require that the last read node is connected to the rest of the list; there is no need to verify the values of other traversed nodes, since the operation no longer depends on them. In this paper, we introduce a new framework for reasoning about correctness of concurrent data structures, which replaces validation of the entire read-set with more general conditions: instead of verifying that all read shared variables still hold the values read from them, we verify abstract conditions over the variables. These are captured by our new notion of *base conditions*.

Roughly speaking, a *base condition* of a read-only operation at time t, is a predicate over shared variables, (typically ones read by the operation), that determines the state the operation has reached at time t. Base conditions are defined over sequential code. Intuitively, they represent invariants the correctness of the read-only operations relies upon in sequential executions. We show that the correctness of the implementation in a concurrent execution depends on whether these invariants are preserved by the update operations executed concurrently with the read-only ones. We capture this formally through the notion of a snapshot condition, which requires that each read-only operation has a *consistent snapshot* of some base condition at any given time. In the linked list example – it does not hurt to see old values in one section of the list and new ones in another section, as long as we read every next pointer consistently with the element it points to. Indeed, this is the intuition behind the famous hand-over-hand locking (lock-coupling) approach [22, 4].

Our framework yields a methodology for verifiable SWMR parallelization. In essence, it suffices for programmers to identify base conditions for their sequential data structure's read-only operations. Then, they can transform their sequential code using means such as readers-writer locks or RCU, to ensure that read-only operations see consistent snapshots when run concurrently with updates.

It is worth noting that there is a degree of freedom in defining base conditions. If coarsely defined, they can constitute the validity of the entire read-set, yielding coarse-grained synchronization as in snapshot isolation and transactional memories. Yet using more precise observations based on the data structure's inner workings can lead to fine-grained base conditions and to better concurrency. Our formalism thus applies to solutions ranging from validation of the entire read-set [10], through multi-versioned concurrency control [6], which has read-only operations read a consistent snapshot of their entire read-set, to fine-grained solutions that hold a small number of locks, like hand-over-hand locking.

Overview of Contributions This paper makes several contributions that arise from our observation regarding the key role of base conditions. While obtaining consistent snapshots of base conditions in some sense ensures "correctness" of a data structure, somewhat surprisingly, we observe that it does not suffice for the commonly-used correctness criterion of *linearizability (atomicity)* [15] or even *sequential consistency* [16] (in Appendix E). Rather, it guarantees a correctness notion weaker than linearizability, similar to Lamport's *regularity* semantics for registers, which we extend here for general objects for the first time. In addition, it ensures a property we call *validity*, which specifies that a concurrent execution does not reach local states that are not reachable in sequential ones. Intuitively, this property is needed in order to avoid situations like division by zero during the execution of the operation.

In Section 2 we present a formal model for shared memory data structure implementations and executions, and define correctness criteria. Section 3 presents our methodology for achieving regularity and validity: We formally define the notion of a base condition, as well as the snapshot condition, which links the sequentially-defined base conditions to concurrent executions. We then prove that the snapshot condition implies regularity and validity. We proceed to exemplify our methodology for three standard linked list implementations – two are given in Section 4, and one is deferred to Appendix C due to space limitations. In Section 5, we define an additional condition on update operations, namely, having a *single visible mutation point*, which along with the snapshot condition ensures linearizability. Finally, we note that we see this paper as the first step in an effort to simplify reasoning about fine-grained concurrent implementations. It opens many directions for future research, which we overview in Section 6. Due to space considerations, some formal definitions and proofs are deferred to appendices, as is our result about sequential consistency.

Comparison with Other Approaches The regularity correctness condition was introduced by Lamport [17] for SWMR registers. It was later extended to multi-writer registers by Shao et al. [24]. To the best of our knowledge, the regularity of a data structure as we present in this paper is a new extension of the definition.

Using our methodology, proving correctness relies on defining a base condition for every state in a given sequential implementation. One easy way to do so is to define base conditions that capture the entire read-set, i.e., specify that there is a point in the execution where all shared variables the operation has read hold the values that were previously read from them. But often, such a definition of base conditions is too strict, and spuriously excludes correct concurrent executions. Our definition generalizes it and thus allows for more parallelism in implementations.

Opacity [12] defines a sufficient condition for validity and linearizability, but not a necessary one. It requires that every transaction see a consistent snapshot of all values it reads, i.e., that all these values belong to the same sequentially reachable state. We relax the restriction on shared states by allowing consistent snapshots of base conditions instead of the entire read-set.

Snapshot isolation [5] guarantees that no operation ever sees updates of concurrent operations. This restriction is a special case of the possible snapshot points that our snapshot condition defines, and thus also implies our condition for the entire read-set.

We prove that the single visible mutation condition, along with the snapshot condition, suffices for linearizability. There are mechanisms, for example, transactional memory implementations [10], for which it is easy to see that these two conditions hold for base conditions that capture the entire read-set. Thus, the theorems that we prove imply correctness of such implementations.

In this paper we focus on correctness conditions that can be used for deriving a correct SWMR data structure from a sequential implementation. The implementation itself may rely on known techniques such as locking, RCU [20], pessimistic lock-elision [1], or any combinations of those, such as RCU combined with fine-grained locking [2]. There are several techniques, such as flat-combining [13] and read-write locking [9], that can naturally expand a SWMR implementation into a multi-writer one by adding synchronization among update operations.

Algorithm designers usually prove linearizability of a data structure by identifying a serialization point for every operation, or showing the existence of a specific partial ordering of operations [8]. Our approach simplifies reasoning – all the designer needs to do now is identify a base condition for every state in the existing sequential implementation, and show that it holds under concurrency. This is often easier than finding and proving serialization points, as we exemplify. Another approach that simplifies verifiable parallelization is to re-write the data structure using primitives that guarantee linearizability [7]. Whereas the latter focuses on non-blocking multi-writer data structure implementations using their primitive, our work is focused on single-writer solutions, and does not restrict the implementation; in particular, we target lock-based implementations as well as non-blocking ones.

2 Model and Correctness Definitions

We consider a shared memory model where each process performs a sequence of operations on shared data structures. The data structures are implemented using a set $X = \{x_1, x_2, ...\}$ of shared variables. The shared variables support atomic read and write operations (i.e., are atomic registers), and are used to implement more complex data structures. The values in the x_i's are taken from some domain $\mathcal{V}$.

2.1 Data Structures and Sequential Executions

A *data structure implementation* (algorithm) is defined as follows:

- A set of states, $\mathcal{S}$, were a *shared state* $s \in \mathcal{S}$ is a mapping $s : X \rightarrow \mathcal{V}$, assigning values to all shared variables. A set $\mathcal{S}_0 \subseteq \mathcal{S}$ defines *initial states.*
- A set of operations representing methods and their parameters. For example, $find(7)$ is an operation. Each *operation op* is a state machine defined by:
  - A set of local states $\mathcal{L}_{op}$, which are usually given as a set of mappings l of values to local variables. For example, for a local state l, $l(y)$ refers to the value of the local variable y in l. $\mathcal{L}_{op}$ contains a special initial local state $\perp \in \mathcal{L}_{op}$.
  - A deterministic transition function $\tau_{op}(\mathcal{L}_{op} \times \mathcal{S}) \rightarrow Steps \times \mathcal{L}_{op} \times \mathcal{S}$ where $step \in Steps$ is a transition label, which can be *invoke*, $a \leftarrow$ *read*(x_i), *write*(x_i,v), or *return*(v) (see Appendix A for more details). Note that there are no atomic read-modify-write steps. Invoke and return steps interact with the application while read and write steps interact with the shared memory.

  We assume that every operation has an isolated state machine, which begins executing from local state $\perp$.

For a transition $\tau(l, s) = \langle step, l', s' \rangle$, l determines the step. If *step* is an invoke, return, or write step, then l' is uniquely defined by l. If *step* is a read step, then l' is defined by l and s, specifically, $read(x_i)$ is determined by $s(x_i)$. Since only write steps can change the content of shared variables, $s = s'$ for invoke, return, and read steps.

For the purpose of our discussion, we assume the entire shared memory is statically allocated. This means that every read step is defined for every shared state in $\mathcal{S}$. One can simulate dynamic allocation in this model by writing to new variables that were not previously used. Memory can be freed by writing a special value, e.g., "invalid", to it.

A state consists of a local state l and a shared state s. By a slight abuse of terminology, in the following, we will often omit either shared or local component of the state if its content is immaterial to the discussion. A *sequential execution of an operation* is an alternating sequence of steps and states with transitions being according to τ. A *sequential execution of a data structure* is a sequence of operation executions that begins in an initial state; see Appendix A for a formal definition. A *read-only operation* is an operation that does not perform write steps in any execution. All other operations are *update operations.*

A state is *sequentially reachable* if it is reachable in some sequential execution of a data structure. By definition, every initial state is sequentially reachable. The *post-state* of an invocation of operation o in execution μ is the shared state of the data structure after o's return step in μ; the *pre-state* is the shared state before o's invoke step. Recall that read-only operations do not change the shared state and execution of update operations is serial. Therefore, every pre-state and post-state of an update operation in μ is sequentially reachable. A state st' is sequentially reachable from a state st if there exists a sequential execution fragment that starts at st and ends at st'.

In order to simplify the discussion of initialization, we assume that every execution begins with a dummy (initializing) update operation that does not overlap any other operation.

2.2 Correctness Conditions for Concurrent Data Structures

A *concurrent execution fragment of a data structure* is a sequence of interleaved states and steps of different operations, where state consists of a set of local states $\{l_i, ..., l_j\}$ and a shared state s_k, where every l_i is a local state of a pending operation. A *concurrent execution of a data structure* is a concurrent execution fragment of a data structure that starts from an initial shared state. Note that a sequential execution is a special case of concurrent execution. An example of a concurrent execution is detailed in Appendix A.

A *single-writer multiple-reader (SWMR) execution* is one in which update operations are not interleaved; read-only operations may interleave with other read-only operations and with update operations. In the remainder of this paper we discuss only SWMR executions.

For an execution σ of data structure ds, the *history* of σ, denoted H_σ, is the subsequence of σ consisting of the invoke and return steps in σ (with their respective return values). For a history H_σ, *complete(H_σ)* is the subsequence obtained by removing pending operations, i.e., operations with no return step, from H_σ. A history is *sequential* if it begins with an invoke step and consists of an alternating sequence of invoke and return steps.

A data structure's correctness in sequential executions is defined using a *sequential specification*, which is a set of its allowed sequential histories.

Given a correct sequential data structure, we need to address two aspects when defining its correctness in concurrent executions. As observed in the definition of opacity [12] for memory transactions, it is not enough to ensure serialization of completed operations, we must also prevent operations from reaching undefined states along the way. The first aspect relates to the data structure's external behavior, as reflected in method invocations and responses (i.e., histories):

Linearizability and Regularity A history H_σ is *linearizable* [15] if there exists H'_σ that can be created by adding zero or more return steps to H_σ, and there is a sequential permutation π of complete(H'_σ), such that: (1) π belongs to the sequential specification of ds; and (2) every pair of operations that are not interleaved in σ, appear in the same order in σ and in π. A data structure ds is *linearizable*, also called *atomic*, if for every execution σ of ds, H_σ is linearizable.

We next extend Lamport's regular register definition [17] for SWMR data structures (we do not discuss regularity for MWMR executions, which can be defined similarly to [24]). A data structure ds is *regular* if for every execution σ of ds, and every read-only operation $ro \in H_\sigma$, if we omit all other read-only operations from H_σ, then the resulting history is linearizable.

Validity The second correctness aspect is ruling out bad cases like division by zero or access to uninitialized data. It is formally captured by the following notion of *validity*: A data structure is *valid* if every local state reached in an execution of one of its operations is sequentially reachable.

3 Base Conditions and Regularity

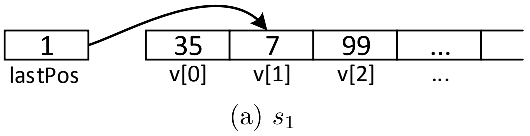
3.1 Base Conditions and Consistent Snapshots

Intuitively, a base condition establishes some link between the local state an operation reaches and shared variables the operation has read before reaching this state. It is given as a predicate Φ over shared variable assignments. Formally:

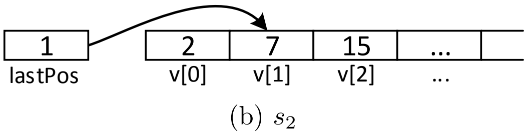
Definition 1 (Base Condition). *Let l be a local state of an operation op. A predicate Φ over shared variables is a* base condition *for l if every sequential execution of op starting from a shared state s such that $\Phi(s) = true$, reaches l.*

For completeness, we define a base condition for $step_i$ in an execution μ to be a base condition of the local state that precedes $step_i$ in μ.

Consider a data structure consisting of an array of elements v and a variable $lastPos$, whose last element is read by the function *readLast*. An example of an execution fragment of *readLast* that starts from s_1 (Figure 1) and the corresponding base conditions appears in Algorithm 1. The *readLast* operation needs the value it reads from $v[tmp]$ to be consistent with the value of $lastPos$ that it reads into tmp because if $lastPos$ is newer than $v[tmp]$, then $v[tmp]$ may contain garbage. Appendix B.1 details base conditions for every possible local state of *readLast*.



(a) s_1



(b) s_2

Figure 1: Two shared states satisfying the same base condition $\Phi_3 : lastPos = 1 \wedge v[1] = 7$.

Algorithm 1: The local states and base conditions of readLast when executed from s_1. The shared variable $lastPos$ is the index of the last updated value in array v. See Algorithm 2 for the corresponding update operation.

local state	***base condition***	**Function** *readLast*()
$l_1 : \{\}$	$\Phi_1 : true$	1. $tmp \leftarrow$ ***read***$(lastPos)$
$l_2 : \{tmp = 1\}$	$\Phi_2 : lastPos = 1$	2. $res \leftarrow$ ***read***$(v[tmp])$
$l_3 : \{tmp = 1, res = 7\}$	$\Phi_3 : lastPos = 1 \wedge v[1] = 7$	3. ***return***(res)

The predicate $\Phi_3 : lastPos = 1 \wedge v[1] = 7$ is a base condition of l_3 because l_3 is reachable from any shared state in which $lastPos = 1$ and $v[1] = 7$ (e.g., s_2 in Figure 1), by executing lines 1-2.

We now turn to define consistent snapshots of base conditions, which link a local state with base condition Φ to a shared state s where $\Phi(s)$ holds.

Definition 2 (Consistent Snapshot)**.** *Let μ be a concurrent execution, ro be a read-only operation executed in μ, and Φ_t be a base condition of the local state and step at index t in μ. An execution fragment of ro in μ has a* consistent snapshot *of Φ_t at point t, if there exists a sequentially reachable post-state s in μ, called a* base point of *t, such that: (1) $\Phi_t(s)$ holds; and (2) s is the post-state of either an update operation executed concurrently with ro in μ or of the last update operation that ended before ro's invoke step in μ.*

The possible base points of a read-only operation *ro* are illustrated in Figure 2. Note that together with Definition 1, the existence of a base point s implies that t is reachable from s in all sequential runs starting from s.

We say that a data structure *ds* satisfies the *snapshot condition* if every point t in every execution of every read-only operation *ro* of *ds* has a consistent snapshot of a base condition of t.

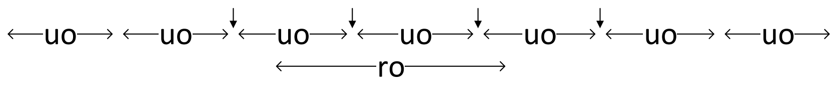


Figure 2: Possible locations of *ro*'s base points.

Algorithm 2: Unlike *writeUnsafe*, *writeSafe* keeps a consistent snapshot of every local state of *readLast*; it guarantees that any concurrent *readLast* operation sees values of $lastPos$ and $v[tmp]$ that occur in the same sequentially reachable post-state. It also has a single visible mutation point (as defined in Section 5), and therefore linearizability is established.

```
Function writeSafe(val)
    $i \leftarrow$ read$(lastPos)$
    write$(v[i+1], val)$
    write$(lastPos, i+1)$
```

```
Function writeUnsafe(val)
    $i \leftarrow$ read$(lastPos)$
    write$(lastPos, i+1)$
    write$(v[i+1], val)$
```

In Algorithm 2 we see two versions of an update operation, *writeSafe* guarantees a consistent snapshot for every local state of *readLast* (Algorithm 1), and *writeUnsafe* does not. As shown in Appendix B.2, *writeUnsafe* can cause a concurrent *readLast* operation interleaved between its two write steps to see values of $lastPos$ and $v[lastPos]$ that do not satisfy *readLast*'s return step's base condition, and to return an uninitialized value.

3.2 Deriving Validity and Regularity from Consistent Snapshots of Base Conditions

We start by proving that the snapshot condition implies validity.

Theorem 1 (Validity)**.** *If a data structure ds satisfies the snapshot condition then ds is valid.*

Proof. In order to prove that ds is valid, we need to prove that for every execution μ of ds, for any operation $op \in \mu$ of ds, every local state is sequentially reachable. If μ is a sequential execution then the claim holds. If op is an update operation, since μ is a SWMR execution then every update operation is executed sequentially starting from a sequentially reachable post-state, thus every local state of op is sequentially reachable. Now we prove for op that is a read-only operation in concurrent execution μ. Given that the data structure satisfies the snapshot condition, every local state l of every read-only operation in μ has a base point s_{base}. In order to show that l is sequentially reachable, we build a sequential execution μ' that starts from the same initial state as μ and consists of the same update operations that appear in μ until s_{base}. Then we add a sequential execution of op. Since s_{base} is a base point of l, l is reached in μ' and therefore is sequentially reachable. ☐

We now prove that the snapshot condition implies regularity.

Lemma 1. *Let μ be a concurrent execution of a data structure ds. Let ro be a read-only operation of ds executed in μ, which returns v. If ds satisfies the snapshot condition then there exists a sequentially reachable shared state s in μ such that: (1) s is the post-state of some update operation that is either concurrent with ro or is the last before ro is invoked; and (2) when executing ro from s, its return value is equal to v.*

Proof. Let l be the local state that precedes ro's return step. Since τ is deterministic, its return value v is fully determined by l, and every execution of ro that reaches l returns v. Given that ds satisfies the snapshot condition, l has a consistent snapshot of some base condition Φ. Let s denote a base point of l and Φ in μ. By the definition of base point, the shared state s is the post-state of some update operation that is either concurrent with ro or is the last before ro is invoked, and $\Phi(s)$ is true. By the definition of base condition Φ, we get that l is reached in ro's sequential execution from s, that is, when ro is sequentially executed from s, its return value is v. ☐

Theorem 2 (Regularity)**.** *If a data structure ds satisfies the snapshot condition then ds is regular.*

Proof. In order to prove that ds is regular, we need to show that for every concurrent execution μ of ds with history H_μ, for any read-only operation $ro \in H_\mu$, if we omit all other read-only operations from H_μ, the resulting history H_μ^{ro} is linearizable. Recall that update operations are executed sequentially.

If μ includes only update operations then μ vacuously satisfies the condition. Otherwise, let ro be a read-only operation in μ. If ro is pending in μ, we build a sequential history by removing ro's invocation from H_μ^{ro}, which is allowed by the definition of linearizability.

Consider now a read-only operation ro that returns in μ. Since every local state of ro has a consistent snapshot in μ, by Lemma 1, we get that there is a shared state s in μ from which ro's sequential execution returns the same value as in μ, and s is the post-state of some update operation that is either concurrent with ro or is the last before ro is invoked. We build a sequential execution μ_{seq}^{ro} from the sequence of update operations in μ with ro added at point s. Then μ_{seq}^{ro} is a sequential execution of ds, which belongs to the sequential specification. Every pair of operations that are not interleaved in μ appear in the same order in μ_{seq}^{ro}. Therefore, H_μ^{ro} is linearizable. ☐

4 Using Our Methodology

We now demonstrate the simplicity of using our methodology. Based on Theorems 1 and 2 above, the proof for correctness of a data structure (such as a linked list) becomes almost trivial. We look at three linked list implementations: Algorithm 3, which assumes managed memory (i.e., automatic garbage collection), Algorithm 4, which uses RCU methodology, and an algorithm based on hand-over-hand locking (deferred to Appendix C.2). For each of the algorithms, we first prove that the listed predicates are indeed base conditions, and next we prove that each algorithm satisfies the snapshot condition. By doing so, and based on Theorems 1 and 2, we get that the algorithms satisfy both validity and regularity.

Consider a linked list node n. Here, $head \overset{*}{\Rightarrow} n$ denotes that there is a set of shared variables $\{head, n_1, ..., n_k\}$ such that $head.next = n_1 \wedge n_1.next = n_2 \wedge ... \wedge n_k = n$, i.e., that there exists some path from the shared variable $head$ to n. Note that n is the only variable associated with a specific read value. We next prove that this defines base conditions for Algorithm 3.

Lemma 2. *In Algorithm 3, Φ_i defined therein is a base condition of the i-th step of* readLast.

Proof. For Φ_1 the claim is vacuously true. For Φ_2, let l be a local state where *readLast* is about to perform the second read step in *readLast*'s code, meaning that $l(next) \neq \perp$. Note that in this local state both local variables n and $next$ hold the same value. Let s be a shared state in which $head \overset{*}{\Rightarrow} l(n)$. Every sequential execution from s iterates over the list until it reaches $l(n)$, hence the same local state where $n = l(n)$ and $next = l(n)$ is reached.

For Φ_3, Let l be a local state where *readLast* has exited the while loop, hence $l(n).next = \perp$. Let s be a shared state such that $head \overset{*}{\Rightarrow} l(n)$. Since $l(n)$ is reachable from $head$ and $l(n).next = \perp$, every sequential execution starting from s exits the while loop and reaches a local state where $n = l(n)$ and $next = \perp$. □

Lemma 3. *In Algorithm 3, if a node n is read during concurrent execution μ of* readLast*, then there is a state where the shared state is s in μ such that n is reachable from* head *in s and ro is pending.*

Proof. A shared variable is read only by a read step, and a read step does not start or terminate an operation. Therefore, if n is read in operation *readLast* from a shared state s, then s exists concurrently with *readLast*. The operation *readLast* starts by reading *head*, and it reaches n. Assume by contradiction that there is no concurrent shared state in μ in which n is reachable from *head*.

Thus, for *readLast* to read n, n must be linked to some node n' at some point in μ. If n was already connected (or added) to the list while n' is still reachable from the head, then there exists a state where n is reachable from the head contradicting the assumption. Otherwise, n needs to be added as the next node of n' at some point in μ after n' is already detached from the list. Nodes are only added via *insertLast* operation which is not executed concurrently with *remove* operation. This means nodes cannot be added to detached elements of the list. A contradiction. □

Algorithm 3: A linked list implementation in a memory-managed environment. For simplicity, we do not deal with boundary cases: we assume that a node can be found in the list prior to its deletion, and that there is a dummy head node.

```
Function remove(n)
    $p \leftarrow \perp$
    $next \leftarrow$ read($head.next$)
    while $next \neq n$
        $p \leftarrow next$
        $next \leftarrow$ read($p.next$)
    write($p.next$, $n.next$)
Function insertLast(n)
    $last \leftarrow readLast()$
    write($last.next$, $n$)
```

```
readLast's base conditions:

$\Phi_1 : true$

$\Phi_2 : head \overset{*}{\Rightarrow} n$
$\Phi_3 : head \overset{*}{\Rightarrow} n$
```

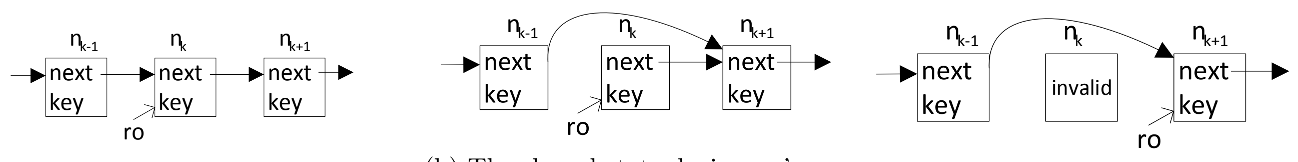
```
Function readLast()
    $n \leftarrow \perp$
    $next \leftarrow$ read($head.next$)
    while $next \neq \perp$
        $n \leftarrow next$
        $next \leftarrow$ read($n.next$)
    return($n$)
```

The following lemma, combined with Theorem 2 above, guarantees that Algorithm 3 satisfies regularity.

Lemma 4. *Every local state of* readLast *in Algorithm 3 has a base point.*

Proof. Let μ be a concurrent execution that contains a *readLast* operation. We now prove that every local state in *readLast* has a consistent snapshot of a base condition. By Lemma 2, the result will follow.

The claim is vacuously true for Φ_1. We now prove for Φ_2 and Φ_3 : $head \overset{*}{\Rightarrow} n$. By Lemma 3 we get that there is a shared state s where *readLast* is pending and satisfies $head \overset{*}{\Rightarrow} n$. Note that n's next field is included in s as part of n's value. Since both update operations - *remove* and *insertLast*, have a single write step, every shared state is a post-state of an update operation. Specifically this means that s is a sequentially reachable post-state, and because *readLast* is pending, s is one of the possible base points of *readLast*. □



(a) *rcuRemove(n_k)*'s pre-state. (b) The shared state during n_k's removal. (c) *rcuRemove(n_k)*'s post-state.

Figure 3: Shared states in a concurrent execution consisting of *rcuRemove(n_k)* and *rcuReadLast* (*ro*).

RCU Read-copy-update [20] is a synchronization strategy that aims to reduce read operations' synchronization overhead as much as possible, while risking a high synchronization overhead for update operations. The idea is that only update operations require locks, and the writes mutate the data structure in a way that ensures that concurrent readers always see a consistent view. Additionally, writers do not free data while it is used by readers. Note that RCU does not allow write-write concurrency.

RCU is commonly used via primitives that resemble readers-writer locks [3]: *rcuReadLock* and *rcuReadUnLock*. There are other primitives that encapsulate list traversal, but we do not use them in our example since we wish to illustrate the general approach. Instead, we use primitives that are commonly used for creating RCU-protected non-list data structures (such as arrays and trees): *rcuWrite(p, v)* (originally called *rcuAssignPointer*), and *rcuRead(p)* (originally called *rcuDereference*) [19].

Algorithm 4: An RCU linked list implementation. *rcuInsertLast*, not listed, is identical to *insertLast* in Algorithm 3.

```
Function rcuRemove(n)
    $p \leftarrow \perp$
    $next \leftarrow$ read($head.next$)
    while $next \neq n$
        $p \leftarrow next$
        $next \leftarrow$ read($p.next$)
    rcuWrite($p.next, n.next$)
    rcuWaitForReaders()
    invalidate($n$)
```

```
rcuReadLast's
base conditions:

    $\Phi_1$ : true

    $\Phi_2$ : $head \overset{*}{\Rightarrow} n$

    $\Phi_3$ : $head \overset{*}{\Rightarrow} n$
```

```
Function rcuReadLast()
    rcuReadLock()
    $n \leftarrow \perp$
    $next \leftarrow$ rcuRead($head.next$)
    while $next \neq \perp$
        $n \leftarrow next$
        $next \leftarrow$ rcuRead($n.next$)
    rcuReadUnlock()
    return($n$)
```

In Algorithm 4, $rcuWrite$ is a write step that changes the next pointer of n's predecessor, and it occurs between the shared states (a) and (b) in Figure 3. The invalidation of n takes place once all read-only operations that use n no longer hold a reference to it, as guaranteed by $rcuWaitForReaders()$. The latter happens between the shared states of (b) and (c). The *rcuReadLast* operation holds at most a single reference to list node at a given time, and our base condition links *head* to it. We see in Figure 3 that invalid nodes are unreachable from *head* in sequentially reachable post-states. Thus, the base condition $head \overset{*}{\Rightarrow} n$ implies that *ro* never holds a pointer to an invalid node.

The correctness of the base conditions annotated in Algorithm 4 follows the same reasoning as Lemma 2, and hence we omit it here. In Appendix C.1 we prove that Algorithm 4 satisfies the snapshot condition. Theorems 1 and 2 proven in Section 3.2 shows that validity and regularity is guaranteed in Algorithm 4, due to the fact that the update operation maintains consistent snapshots of the base conditions that we identified.

5 Visible Mutation Points and Linearizability

We first show that the snapshot condition is insufficient for linearizability. In Figure 4 we show an example of a concurrent execution where two read-only operations ro_1 and ro_2 are executed sequentially, and both have consistent snapshots. The first operation ro_1 reads the shared variable *first name* and returns Joe, and the second operations ro_2 reads the shared variable *surname* and returns Doe. An update operation *uo* updates the data structure concurrently, using multiple write steps. The return step of ro_1 is based on the post-state of *uo*, whereas ro_2's return step is based on the pre-state of *uo*. There is no sequential execution of the operations where ro_1 returns Joe and ro_2 returns Doe.

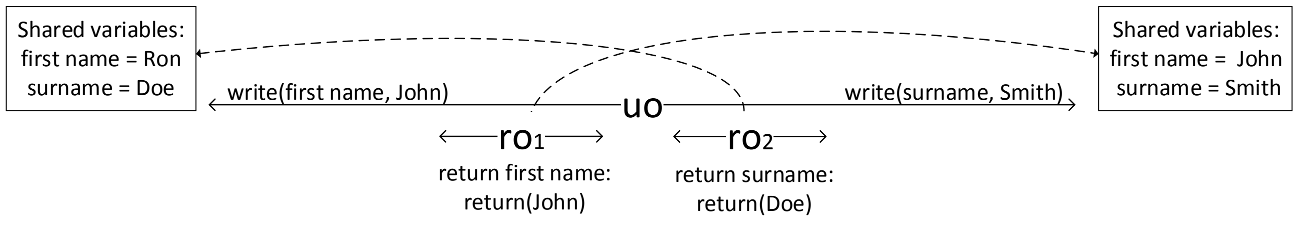


Figure 4: Every local state of ro_1 and ro_2 has a base point, and still the data structure is not linearizable. Here we see an execution which history is not linearizable. If ro_1 and ro_2 belong to the same process, then the history is not even sequentially consistent (see Appendix E).

Thus, an additional condition is required for linearizability. We suggest a condition related to the number of *visible mutation points* an update operation has. Intuitively, a visible mutation point of an update operation is a write step that writes to a shared variable that might be read by concurrent operations. A more formed definition ensues.

Let α be an execution fragment of *op* starting from a shared state s. We define α^t as the prefix of α consisting of t steps of *op*, and we denote by $steps_{op}(\alpha)$ the subsequence of α consisting of the steps of *op* in α. We say that α^t and α^{t-1} are *indistinguishable* to a concurrent read-only operation ro if for every concurrent execution μ_t starting from s and consisting only of steps of ro and α^t, and concurrent execution μ_{t-1} starting from s and consisting only of steps of ro and α^{t-1}, $steps_{ro}(\mu_t) = steps_{ro}(\mu_{t-1})$. In other words, ro is unaffected by the t'th step of *op*.

If α^t and α^{t-1} are indistinguishable to every possible concurrent read-only operation, then point t in α^t is a *silent point*. A point that is not silent is a *visible mutation point*.

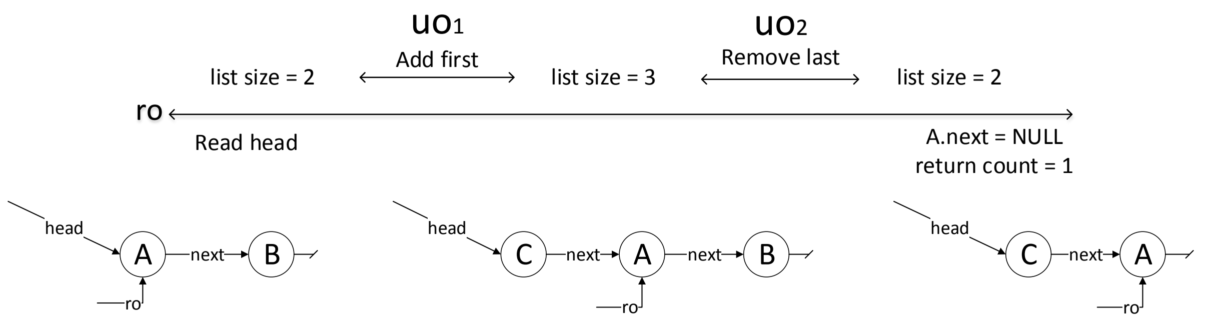
Definition 3 (Single visible mutation condition)**.** *A data structure ds satisfies the single visible mutation condition if each of its update operations has a single visible mutation point.*

If a data structure satisfies the single mutation condition and not the snapshot condition, it is not necessarily linearizable. Intuitively, the reason is that different update operations can affect the same read-only operation. For example, in Figure 5 we see two sequential single visible mutation point operations, and a concurrent read-only operation *ro* that counts the number of elements in a list. Since *ro* only sees one element of the list, it returns 1, even though there is no shared state in which the list is of size 1. Thus, the execution is not linearizable or even regular.

Intuitively, if a data structure *ds* satisfies the single visible mutation point condition, then all of its shared states are sequentially reachable post-states. If *ds* also satisfies the snapshot condition, then the visible mutation point condition guarantees that the order between the base points of non-interleaved read-only operations is equal to the real time order between those operations. Note that in Algorithms 3 and 4 each of the remove operations has a single visible mutation point, which is the step that writes to *p.next*. The invalidation step in Algorithm 4 changes a node that is not reachable and is therefore a silent step. Thus, from Theorem 3, these implementations are linearizable.

The following Theorem is proven in Appendix D.

Theorem 3 (Linearizability of a data structure)**.** *If ds is a data structure that satisfies the single visible mutation condition and the snapshot condition then ds is linearizable.*



(a) The initial shared state. (b) The post-state of uo_1. (c) The post-state of uo_2.

Figure 5: Every update operation has a single visible mutation point, but the history of the execution is not linearizable.

6 Conclusions and Future Directions

We introduced a new framework for reasoning about correctness of data structures in concurrent executions, which facilitates the process of verifiable parallelization of legacy code. Our methodology consists of identifying base conditions in sequential code, and ensuring consistent snapshots of these conditions under concurrency. This yields two essential correctness aspects in concurrent executions – the internal behaviour of the concurrent code, which we call validity, and the external behaviour, in this case regularity, which we have generalized here for data structures. Linearizability is guaranteed if the implementation further satisfies the single visible mutation point condition.

We believe that this paper is only the tip of the iceberg, and that many interesting connections can be made using the observations we have presented. For a start, a natural expansion of our work would be to consider also multi-writer data structures. Another interesting direction to pursue is to use our methodology for proving the correctness of more complex data structures than the linked lists in our examples.

Currently, using our methodology involves manually identifying base conditions. It would be interesting to create tools for suggesting a base condition for each local state. One possible approach is to use a dynamic tool that identifies likely program invariants, as in [11], and suggests them as base conditions. Alternatively, a static analysis tool can suggest base conditions, for example by iteratively accumulating read shared variables and omitting ones that are no longer used by the following code (i.e., shared variables whose values are no longer reflected in the local state).

Another interesting direction for future work might be to define a synchronization mechanism that uses the base conditions in a way that is both general purpose and fine-grained. A mechanism of this type will use default conservative base conditions, such as verifying consistency of the entire read-set for every local state, or two-phase locking of accessed shared variables. In addition, the mechanism will allow the users to manually define or suggest finer-grained base conditions. This can be used to improve performance and concurrency, by validating the specified base condition instead of the entire read-set, or by releasing locks when the base condition no longer refers to the value read from them.

From a broader perspective, we showed how correctness can be derived from identifying inner relations in a sequential code, (in our case, base conditions), and maintaining those relations in concurrent executions (via consistent snapshots). It may be possible to use similar observations in other models and contexts, for example, looking at inner relations in synchronous protocols, in order to derive conditions that ensure their correctness in asynchronous executions.

Acknowledgements

We thank Naama Kraus, Dahlia Malkhi, Yoram Moses, Dani Shaket, Noam Shalev, and Sasha Spiegelman for helpful comments and suggestions.

References

[1] Y. Afek, A. Matveev, and N. Shavit. Pessimistic software lock-elision. In *Proceedings of the 26th International Conference on Distributed Computing*, DISC'12, pages 297–311, Berlin, Heidelberg, 2012. Springer-Verlag.

[2] M. Arbel and H. Attiya. Concurrent updates with rcu: Search tree as an example. PODC '14, New York, NY, USA, 2014. ACM.

[3] A. Arcangeli, M. Cao, P. E. McKenney, and D. Sarma. Using read-copy-update techniques for system v ipc in the linux 2.5 kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 297–309. USENIX, 2003.

[4] R. Bayer and M. Schkolnick. Readings in database systems. chapter Concurrency of Operations on B-trees, pages 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[7] T. Brown, F. Ellen, and E. Ruppert. Pragmatic primitives for non-blocking data structures. In *PODC*, pages 13–22, 2013.

[8] G. Chockler, N. Lynch, S. Mitra, and J. Tauber. Proving atomicity: An assertional approach. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 152–168, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] P.-J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[12] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[13] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22Nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

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

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[17] L. Lamport. On interprocess communication. part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[18] P. E. McKenney. Selecting locking primitives for parallel programming. *Commun. ACM*, 39(10):75–82, Oct. 1996.

[19] P. E. McKenney. RCU part 3: the RCU API. January 2008.

[20] P. E. McKenney and J. D. Slingwine. Read-copy update: using execution history to solve concurrency problems, parallel and distributed computing and systems, 1998.

[21] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors*, pages 47–14 47–30, 2007. Chapman and Hall/CRC Press.

[22] B. Samadi. B-trees in a system with multiple users. *Inf. Process. Lett.*, 5(4):107–112, 1976.

[23] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[24] C. Shao, J. L. Welch, E. Pierce, and H. Lee. Multiwriter consistency conditions for shared memory registers. *SIAM J. Comput.*, 40(1):28–62, 2011.

[25] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

A Definitions of Steps and Executions

We now detailed the possible steps of an operation. Recall that for operation op, the deterministic transition function τ_{op} is defined as $\tau_{op}(\mathcal{L}_{op} \times \mathcal{S}) \to Steps \times \mathcal{L}_{op} \times \mathcal{S}$. The steps induce the following atomic transitions:

- An *invoke* changes the initial local state $\perp$ into another local state, and does not change the shared state.
- A $write(x_i, v)$ changes the local state and changes the value of shared variable $x_i \in X$ to v.
- A $a \leftarrow read(x_i)$ reads the value of one variable $x_i \in X$ from the shared state and changes the local state accordingly (i.e., stores the value of x_i in a local variable a).
- A $return(v)$ ends the operation by changing the local state to $\perp$ and returning v to the calling process. It does not change the shared state.

A *sequential execution of an operation* from a shared state $s_i \in \mathcal{S}$ is a sequence of transitions of the form:

$$\genfrac{}{}{0pt}{}{\perp}{s_i},\ invoke,\ \genfrac{}{}{0pt}{}{l_1}{s_i},\ step_1,\ \genfrac{}{}{0pt}{}{l_2}{s_{i+1}},\ step_2,\ \dots,\ \genfrac{}{}{0pt}{}{l_k}{s_j},\ return_k,\ \genfrac{}{}{0pt}{}{\perp}{s_j},$$

where $\tau(l_m,\ s_n) = \langle step_m,\ l_{m+1},\ s_{n+1}\rangle$. The first step is invoke, ensuing steps are read or write steps, and the last step is a return step.

A *sequential execution of a data structure* is a (finite or infinite) sequence μ:

$$\mu = \genfrac{}{}{0pt}{}{\perp}{s_1},\ O_1,\ \genfrac{}{}{0pt}{}{\perp}{s_2},\ O_2,\ \dots,$$

where $s_1 \in \mathcal{S}_0$ and every $\genfrac{}{}{0pt}{}{\perp}{s_j},\ O_j,\ \genfrac{}{}{0pt}{}{\perp}{s_{j+1}}$ in μ is a sequential execution of some operation. If μ is finite, it can end after an operation or during an operation. In the latter case, we say that the last operation is *pending* in μ. Note that in a sequential execution there can be at most one pending operation.

We use the same notation for concurrent executions. For example, the following is a concurrent execution fragment that starts from a shared state s_i and invokes two operations: O_A and O_B. The first operation takes a write step, and then O_B takes a read step. We subscript every step and local state with the operation it pertains to.

$$\genfrac{}{}{0pt}{}{\emptyset}{s_i},\ invoke_A(),\ \genfrac{}{}{0pt}{}{\{l_{1,A}\}}{s_i},\ write_A(x_i, v),\ \genfrac{}{}{0pt}{}{\{l_{2,A}\}}{s_{i+1}},\ invoke_B(),\ \genfrac{}{}{0pt}{}{\{l_{2,A},\ l_{1,B}\}}{s_{i+1}},\ a \leftarrow read_B(x_i),\ \genfrac{}{}{0pt}{}{\{l_{2,A}, l_{2,B}\}}{s_{i+1}}.$$

B A Closer Look at ReadLast

B.1 Defining Base Conditions for ReadLast

In Algorithm 5 we see a base condition for every local state that is reached during execution of *readLast*.

Algorithm 5: ReadLast operation. The shared variable *lastPos* is the index of the last updated value in array v. See Algorithm 2 for the corresponding update operation.

```
Shared variables: $lastPos$, $\forall i \in \mathbb{N} : v[i]$
        base condition                                 step
        $\Phi_1 : true$                                   $tmp \leftarrow \textbf{read}(lastPos)$
        $\Phi_2 : lastPos = tmp$                          $res \leftarrow \textbf{read}(v[tmp])$
        $\Phi_3 : lastPos = tmp \wedge v[tmp] = res$      $\textbf{return}(res)$
```

B.2 Satisfying the Snapshot Condition

Let us examine the possible concurrent executions an invocation *ro* of *readLast* (Algorithm 1) and an invocation *uo* of *writeSafe* (Algorithm 2) with parameter 80 starting from s_1 (Figure 1). There are four possible interleavings of write steps of *uo* and read steps of *ro* starting from s_1 shown in Algorithm 6. In each of them, *ro* returns 7, and s_1 is the base point of its last local state.

Algorithm 6: Four interleaved executions of invocation *ro* of *readLast* and invocation *uo* of *writeSafe* that start from s_1.

μ_1 :	μ_2 :	μ_3 :	μ_4 :
$\boldsymbol{read_{ro}}(lastPos)$	$\boldsymbol{read_{ro}}(lastPos)$	$read_{uo}(lastPos)$	$read_{uo}(lastPos)$
$read_{uo}(lastPos)$	$read_{uo}(lastPos)$	$write_{uo}(v[2], 80)$	$write_{uo}(v[2], 80)$
$write_{uo}(v[2], 80)$	$write_{uo}(v[2], 80)$	$\boldsymbol{read_{ro}}(lastPos)$	$\boldsymbol{read_{ro}}(lastPos)$
$\boldsymbol{write_{uo}}(lastPos, 2)$	$read_{ro}(v[1])$	$\boldsymbol{write_{uo}}(lastPos, 2)$	$read_{ro}(v[1])$
$read_{ro}(v[1])$	$return_{ro}(7)$	$read_{ro}(v[1])$	$return_{ro}(7)$
$return_{ro}(7)$	$\boldsymbol{write_{uo}}(lastPos, 2)$	$return_{ro}(7)$	$\boldsymbol{write_{uo}}(lastPos, 2)$

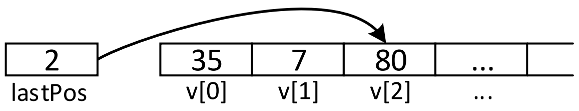


Figure 6: The shared state s'_1. It is the post-state after executing *writeSafe* or *writeUnsafe* from s_1 (Figure 1) with initial value 80.

Now let us examine a concurrent execution consisting of *readLast* and *writeUnsafe* (Algorithm 2), in which *readLast* reads a value from *lastPos* right after *writeUnsafe* writes to it. In Algorithm 7 we see such an execution that starts from s_1. The last local state of *ro* is $l'_3 = \{tmp = 2, res = 99\}$. Neither s_1 and s'_1 satisfies $\Phi'_3 : lastPos = 2 \wedge v[2] = 99$, meaning that l'_3 does not have a consistent snapshot of Φ'_3.

Algorithm 7: A possible concurrent execution consisting of *readLast* and *writeUnsafe*, starting from s_1.

```
$read_{seq}(lastPos)$
$\boldsymbol{write_{seq}}(lastPos, 2)$
$\boldsymbol{read_{ro}}(lastPos)$
$read_{ro}(v[2])$
$return_{ro}(99)$
$write_{seq}(v[2], 80)$
```

Below we show that this is not an artifact of our choice of a base condition – we prove that for every base condition Φ'_3 of l'_3, both $\Phi'_3(s_1)$ and $\Phi'_3(s'_1)$ are false.

Lemma 5. *The snapshot condition does not hold in a data structure that has both writeUnsafe and readLast operations.*

Proof. Given the execution of Algorithm 7 that starts from the shared state s_1 and ends in shared state s'_1 depicted in Figure 6, we assume by contradiction that there is such a base condition of $l'_3 = \{tmp = 2, res = 99\}$ that is satisfied by s_1 or s'_1. By the definition of base condition, if we execute *readLast* sequentially from a shared state that satisfies its base condition, we reach l'_3. But if we execute *readLast* from s_1 we reach $l_3 : \{tmp = 1, res = 7\}$ and if we execute from s'_1 we reach $l''_3 : \{tmp = 2, res = 80\}$. A contradiction. $\square$

C Proving the Snapshot Condition for the Linked List Examples

In the following sections we prove that every local state of *rcuReadLast* and *hohReadLast* have a base point, meaning that Algorithms 4 and 8 satisfies the snapshot condition.

C.1 RcuReadLast

Lemma 6. *In Algorithm 4, if a node n is read during concurrent execution μ of* rcuReadLast*, then there is a state where the shared state is s in μ such that n is reachable from* head *in s and ro is pending.*

The proof of Lemma 6 follows the same reasoning as Lemma 3, and hence we omit it here.

Lemma 7. *Every local state of* rcuReadLast *in Algorithm 4 has a base point.*

Proof. Every read step is encapsulated by *rcuRead*, and is surrounded by *rcuReadLock* and *rcuReadUnlock*. These calls guarantee that as long as the reader holds a reference to the value it read using *rcuRead*, the value cannot be changed by an update operation, since the update operation's write step uses *rcuWrite*. The writer waits for all readers to forget a node before invalidating it, and invalidates it only after it is not reachable. Therefore, it is guaranteed that every node that is read is valid. In addition, Lemma 6 guarantees that for every valid node that is read there is a concurrent shared state s in μ that satisfies the base conditions of *rcuReadLast*. Since the invalidation is not visible to the readers, the post-state of *rcuRemove* and the shared state after *rcuWaitForReaders* are indistinguishable to the readers. Therefore, every shared state is equivalent from the reader perspective to a sequentially reachable post-state. In conclusion, every local state l has a sequentially reachable post-state that is equivalent to s and therefore is a base point of l. ☐

C.2 Locking and HohReadLast

In *hand-over-hand locking*, a data structure is traversed by holding a lock to the next node in the traversal before unlocking the previous one.

In Algorithm 8 we give a linked list implementation using hand-over-hand locking. The locks used therein are readers-writer locks [18], where write locks are exclusive and multiple threads can obtain read locks concurrently. We define a lock for every shared variable $x_i \in X$, and extend the model with $lock(x_i)$ and $unlock(\{x_{i_1}, x_{i_2}, ...\})$ steps. The correctness of the base conditions annotated in Algorithm 8 follows the same reasoning as Lemma 2, and hence we omit it here. The reachable post-states in Figure 3 are (a) and (c). State (b) does not occur in this implementation since *ro* cannot access n concurrently with an update operation that holds n's lock. In the following lemma we prove that Algorithm 8 satisfies the snapshot condition.

Algorithm 8: A linked list implementation using hand-over-hand locking.

```
Function hohRemove(n)
    $p \leftarrow \bot$
    lock($head.next$)
    $next \leftarrow$ read($head.next$)
    while $next \neq n$
        $p \leftarrow next$
        lock($p.next$)
        unlock($p$)
        $next \leftarrow$ read($p.next$)
    write($p.next, n.next$)
    lock($n$)
    invalidate($n$)
    unlock($n$, $p$)
```

```
hohReadLast's
base conditions:        Function hohReadLast()
                            $n \leftarrow \bot$
                            lock($head.next$)
$\Phi_1 : true$             $next \leftarrow$ read($head.next$)
                            while $next \neq \bot$
                                $n \leftarrow next$
                                lock($n.next$)
$\Phi_2 : head \overset{*}{\Rightarrow} n$      $next \leftarrow$ read($n.next$)
                                unlock($n$)
                            unlock($next$)
$\Phi_3 : head \overset{*}{\Rightarrow} n$  return($n$)
```

Lemma 8. *In Algorithm 8, if a node n is read during concurrent execution μ of* hohReadLast*, then there is a state where the shared state is s in μ such that n is reachable from* head *in s and ro is pending.*

The proof of Lemma 8 follows the same reasoning as Lemma 3, and hence we omit it here.

Lemma 9. *Every local state of* hohReadLast *in Algorithm 8 has a base point.*

Proof. In *hohReadLast*, the reader reads a node only after locking it. In addition, the writer invalidate a node only after he locked it. Therefore, the reader only sees valid nodes. From Lemma 8 we get that for every valid node that is read there is a concurrent shared state s in μ that satisfies the base conditions of *hohReadLast*. Since the invalidation is not visible to the readers, the post-state of *hohRemove* and the shared state after the write step are indistinguishable to the readers. Therefore, every shared state is equivalent from the reader perspective to a sequentially reachable shared state. In conclusion, every local state l has a sequentially reachable post-state that is equivalent to s and therefore is a base point of l. ☐

D Proof of Linearizability

The following lemmas show that when a data structure satisfies both the snapshot and the single visible mutation conditions, it is possible to linearize every execution by selecting the base point of the return step of every read-only operations as linearization point, and the visible mutation point as the linearization point of every update operation. Therefore, every execution is linearizable.

For steps or states n and m, the notation $n <_\mu m$ denotes that n appears before m in μ, and $n \leq_\mu m$ if $n <_\mu m$ or n occurred at the same time in μ as m.

Lemma 10. *Let μ be a concurrent execution of a ds that satisfies the snapshot and single visible mutation conditions. Let ro_1 and ro_2 be two non-interleaved read-only operations executed in μ. Let s_1 be the latest base point of the return step of ro_1 in μ, and s_2 be the earliest base point of the return step of ro_2 in μ. If ro_1 is executed before ro_2 in μ then either $s_1 <_\mu s_2$ or $s1$ and s_2 are base points of both return steps.*

Proof.

Let $return_1$ be the return step of ro_1 in μ and $invoke_2$ be the invoke step of ro_2 in μ. Given that ro_1 and ro_2 are not interleaved in μ and ro_1 executed before ro_2, we get that $return_1 <_\mu invoke_2$. Let uo be the last update operation that executed before or concurrently with ro_1 in μ. Such exists since we assume a dummy initialization update operation ro is preceded by at least one update operation.

Let $post_{uo}$ be the post-state of uo in μ, and pre_{ro_2} the pre-state of ro_2 in μ. If $post_{uo} \leq_\mu pre_{ro_2}$, since $s_1 \leq_\mu post_{uo}$ and $pre_{ro_2} \leq_\mu s_2$ we get that $s_1 \leq_\mu s_2$ and the claim holds. Else, $pre_{ro_2} <_\mu post_{uo}$. This means that uo is interleaved with both ro_1 and ro_2 in μ. Assume that $s_2 <_\mu s_1$ (otherwise the claim holds). Let mp_{uo} be the visible mutation point of uo in μ.

Case 1: If $return_1 <_\mu mp_{uo}$ then since $s_1 <_\mu return_1$ we get that $s_2 <_\mu s_1 <_\mu mp_{uo}$ thus s_2 is also a base point of $return_1$ and the claim holds.

Case 2: If $mp_{uo} <_\mu return_1$ then since $return_1 <_\mu invoke_2$ we get that $mp_{uo} <_\mu invoke_2$. Since uo has single visible mutation point, all the values seen by ro_2 belong to a shared state s that is found after mp_{uo}. Thus $s_2 <_\mu s_1 <_\mu mp_{uo} <_\mu s$. This means that uo did not change any value that is found in s_2 and belong to ro_2's return step's consistent snapshot. Therefore all those values can be found in s_1 as well and we get that s_1 is a base point of ro_2's return step. ☐

Lemma 11. *Let μ be a concurrent execution such that: (1) μ starts from a sequentially reachable post-state s; and (2) every update operation in μ is a single visible mutation point operation; and (3) every local state of every read-only operation in μ has a base point.*

Then there is a sequential execution μ_{seq} such that: (1) μ_{seq} and μ contain the same operations; and (2) every pair of non-interleaved operations in μ appear in the same order in μ_{seq} and in μ.

Proof. We build a sequential execution μ_{seq} in the following way: (1) μ_{seq} starts from the same shared state s as μ. It is given that s is sequentially reachable . (2) All update operations in μ appear in the same order in μ_{seq}. Since μ is execution, there are no interleaved update operations. (3) Every read-only operation ro in μ is executed in μ_{seq} right after the first post-state that is a base point of the last local state of ro (i.e., the local state that determines ro's return value). (4) Every pair of read-only operations that are not interleaved in μ appear in μ_{seq} in the same order they appear in μ. If two non-interleaved read-only operations are executed in μ_{seq} after the same post-state, then the order of their execution in μ_{seq} is identical to their execution order in μ. Lemma 10 guarantees that such sequential ordering exists between any pair of non-interleaved read-only operations. (5) The order of read-only operation that are interleaved in μ is arbitrary. Meaning that two read-only operations that are executed after the same post-state and are interleaved in μ, are executed in μ_{seq} in some sequential order.

The execution order of every pair of non-interleaved operations in μ_{seq} is identical to their execution order in μ. Since only update operations can change the shared state and their sequential order is the same in both operations, every update operation is executed in μ_{seq} from the same shared state as in μ. By the definitions of consistent snapshot and base condition we get that every read-only operation in μ_{seq} returns the same value in μ_{seq} as in μ – ro is executed from a shared state that is a base point of its last local state, and the last local state determines ro's return value, thus the return value of ro in both executions is identical since the last local state is identical. In conclusion, every operation's execution that is found in μ found in μ_{seq} as well. □

Theorem 3. *[Linearizability of a data structure] If ds is a data structure that satisfies the single visible mutation condition and the snapshot condition then ds is linearizable.*

Proof. Let μ be a concurrent execution of ds. By Lemma 11 (see Appendix D) we get that there is a sequential execution μ_{seq}, such that $H_{\mu_{seq}}$ is a permutation of complete(H_μ) that belongs to the sequential specification of ds and keeps the order of non-interleaved operations of μ. Thus ds is linearizable. □

E Sequential Consistency

Some systems use the correctness criterion of sequential consistency [16], which relaxes linearizability by not requiring *real time order* (RTO) between operations of different processes. A history H_σ is *sequentially consistent* if there exists H'_σ that can be created by adding zero or more return steps to σ, and there is a sequential permutation π of complete(H'_σ), such that the order of operations that belong to the same process in σ is preserved in π, and π belongs to the sequential specification of ds. A data structure ds is sequentially consistent if for every execution σ, H_σ is sequentially consistent.

Note that sequential consistency and regularity are incomparable: Regularity does not impose RTO on read-only operations even if they belong to the same process, while in sequential consistency, the RTO of read-only operations of the same process is preserved. On the other hand, regularity enforces the RTO between an update operation and every other operation, while sequential consistency allows re-ordering of operations executed by different processes.

In Section 3.1 we defined the snapshot condition, which restricts the choice of base points so as to satisfy RTO. We then proved in Section 3.2 that any data structure that satisfies this condition is regular. In order to satisfy sequential consistency, we now show that a relaxed condition, which does not limit the possible locations of the base point, is sufficient.

Definition 4 (Loose Consistent Snapshot)**.** *Let μ be a concurrent execution, ro be a read-only operation executed in μ and Φ_t be a base condition of the local state and step at index t in μ. Point t in an execution fragment of ro in μ has a* loose consistent snapshot *of base condition Φ_t, if there exists a sequentially reachable post-state s in μ, called a* loose base point *of t, such that $\Phi_t(s)$ holds and s is a post-state of some update operation in μ.*

We say that a data structure ds satisfies the *loose snapshot condition* if there exist loose base points for all read-only operations of ds, so that if two operations of the same process have different loose base points, the loose base points appear in the execution in the same order as the operations do.

First we prove that the loose snapshot condition implies validity.

Lemma 12. *If a data structure ds satisfies the loose snapshot condition then ds is valid.*

Proof. In order to prove that ds is valid, we need to prove that for every execution μ of ds, for any operation $op \in \mu$ of ds, every local state is sequentially reachable. If μ is a sequential execution then the claim holds. If op is an update operation, since μ is a SWMR execution then every update operation is executed sequentially starting from a sequentially reachable post-state, thus every local state of op is sequentially reachable. Now we prove for op that is a read-only operation in concurrent execution μ. Given that the data structure satisfies the snapshot condition, every local state l of every read-only operation in μ has a loose base point s_{base}. In order to show that l is sequentially reachable, we build a sequential execution μ' that starts from the same initial state as μ and consists of the same update operations that appear in μ until s_{base}. Then we add a sequential execution of op. Since s_{base} is a loose base point of l, l is reached in μ' and therefore is sequentially reachable. □

Now we prove that the loose snapshot condition along with the single visible mutation condition ensures sequential consistency.

Lemma 13. *Let μ be a concurrent execution such that every update operation in μ is a single visible mutation point operation; and there exist loose base points for every read-only operation in μ, so that if two operations of the same process have different loose base points, the loose base points appear in the execution in the same order as the operations do. Then there is a sequential execution μ_{seq} such that μ_{seq} and μ contain the same operations with the same return value; and for every process, all its operations appear in the same order in μ_{seq} and in μ.*

Proof. We build a sequential execution μ_{seq} in the following way: (1) μ_{seq} starts from the same shared state s as μ. (2) All update operations in μ appear in the same order in μ_{seq}. Since μ is a SWMR execution, there are no interleaved update operations. (3) Every read-only operation ro in μ is executed in μ_{seq} from a post-state that is a loose base point of the last local state of ro (i.e., the local state that determines ro's return value). It is given that for operations of the same process, different loose base points appear in the execution in the same order as the operations do. Therefore if there are multiple possibilities for loose base point, the operation is executed from the loose base point according to the that order. Read-only operations of the same process that have the same loose base point are executed from it at the same order in μ_{seq} as in μ. (4) The order of read-only operations that do not belong to the same process and are executed from the same loose base point is arbitrary.

Since only update operations can change the shared state and their sequential order is the same in both executions, every update operation is executed in μ_{seq} from the same shared state as in μ. By the definitions of loose consistent snapshot and base condition we get that every read-only operation in μ_{seq} returns the same value in μ_{seq} as in μ – ro is executed from a shared state that is a base point of its last local state, and the last local state determines ro's return value. □

Theorem 4 (Sequential consistency)**.** *If ds is a data structure that satisfies the single visible mutation condition and the loose snapshot condition then ds is sequentially consistent.*

Proof. Let μ be a concurrent execution of ds. By Lemma 13 we get that there is a sequential execution μ_{seq}, such that $H_{\mu_{seq}}$ is a permutation of complete(H_μ) that belongs to the sequential specification of ds and keeps the RTO of operations that belong to the same process in μ. Thus ds is sequentially consistent. □