# On the Limits of Automated Linearizability Verification for Modern Concurrency Patterns*

Durwasa Chakraborty
Indian Institute of Technology Madras
Department of Computer Science
India
cs24d011@cse.iitm.ac.in

## Abstract

The transition to multicore architectures has made concurrent programming a necessity rather than an optimization. While linearizability provides a precise correctness condition for concurrent objects, automatically verifying fine-grained lock-free algorithms remains challenging due to interleavings, complex atomic primitives, and weak memory behavior.

This report examines the progression from formal specifications of concurrent correctness, to automated reasoning techniques, to mechanized verification of realistic lock-free data structures, and finally to abstractions that support compositional concurrency under relaxed memory models. Through this layered study, we highlight the central tension between expressive concurrent algorithms and the rigor required to prove them correct. The overarching theme is that scalable concurrency demands not only sophisticated synchronization primitives, but verification frameworks and abstractions capable of reasoning across semantic, algorithmic, and memory-model boundaries.

## 1 Introduction

In contemporary computing systems, performance improvements are achieved primarily through the effective utilization of multicore architectures, rather than through continued increases in single-core clock frequency. This architectural shift follows the breakdown of Dennard scaling [2], under which further increases in transistor density could no longer be accompanied by proportional reductions in power consumption and heat dissipation. Consequently, processor design has evolved toward multicore architectures as the principal mechanism for performance scaling.

As concurrency has become ubiquitous, the central challenge has shifted from achieving parallelism to reasoning about correctness in its presence. The nondeterminism inherent in concurrent execution, together with the exponential growth of possible interleavings on multicore systems, renders conventional testing methodologies that involves an array of testing suites inadequate. Establishing correctness therefore requires formal reasoning frameworks capable of expressing and verifying properties such as atomicity, linearizability, and progress.

This report is organized as follows. We begin by recalling the formal definition of linearizability [7] in §2 seminal work by Herlihy and Wing, the canonical correctness condition for concurrent objects.

§3 surveys automated approaches to linearizability, focusing on the work of Victor Vafeiadis [13], which demonstrates how linearization points and interference reasoning can be automated.

We then study Zoo [1],in §4, by Allain Clément and Gabriel Scherer, a framework for verifying concurrent OCaml5 programs using separation logic. Zoo embeds a core fragment of OCaml into Rocq and enables machine-checked proofs of fine-grained concurrent data structures. To understand the practical implications of such verification, we examine a data structure based on multi-word compare-and-swap (kCAS)[3] by Guerraoui et al., §5 originally proposed in practical form by Harris et al. [4]. Unlike single-word CAS, kCAS supports atomic updates to multiple memory locations.

Zoo [1] assumes sequential consistency, whereas real-world Multicore OCaml operates under a relaxed memory model. To address this discrepancy, we study Cosmo [10], in §4, by Glen Mével et al., a concurrent separation logic tailored to OCaml's weak memory semantics. Cosmo reconciles high-level reasoning principles with low-level memory behavior, exposing the additional subtleties introduced by relaxed memory.

Finally, we consider Reagents [11], §7, by Aaron Turon, which proposes a transactional abstraction for composing fine-grained concurrent interactions. Reagents structure lock-free algorithms around explicit commit boundaries, offering a compositional model that resembles transactional memory. Through these sections, ie. from specification, to automation, to mechanization, to memory-model awareness, and ultimately to compositional concurrency the report explores how modern verification techniques engage with increasingly expressive concurrent programming paradigms.

*The use of a prepositional opening, often reads as non-assertive. However, this is deliberate: the document is written in reported speech, and this stylistic choice is adopted from Adam Smith's *The Wealth of Nations*.

# 2 Linearizability: A Correctness Condition for Concurrent Objects[7]

## 2.1 Motivation

Concurrency is now a given rather than a design choice, and shared *concurrent objects* are unavoidable in system construction. The remaining challenge is no longer how to exploit concurrency, but how to ascribe precise meaning to object behavior in the presence of arbitrary interleavings. The challenge is how to exploit concurrency and assign precise meaning to object behavior in the presence of interleavings, that may grow astronomically. In the battle between performance and correctness, the canons of sound engineering practices always should prioritize correctness, thus motivating the study of correctness conditions for concurrent objects.

The fundamental question concurrency raises is: what is the intended behavior of an object when its operations are interleaved in many possible ways? If thread A performs $m$ operations and thread B performs $n$ operations, the number of possible interleavings grows combinatorially, on the order of $\binom{m+n}{m}$. Whether such executions are correct depends on the object's specification.

The term "FIFO queue" alone is imprecise. A proper specification formalizes the data structure's behavior by explicitly characterizing the set of valid input/output traces, rather than relying on informal descriptions. In a classic FIFO, queue, if elements are enqueued as $\langle 1, 2, 3\rangle$, every legal execution must dequeue them as verbatim $\langle 1, 2, 3\rangle$. In contrast, large-scale asynchronous systems such Kafka-based queues typically adopt a different delivery specification(s). For example *at-least-once delivery*, in such a system producing $\langle 1, 2, 3\rangle$ may legally result in consumer-visible traces like $\langle 1, 1, 1, 2, 3\rangle$ or multiple replays of $\langle 1, 2, 3\rangle$ $\langle 1, 2, 3\rangle$. Correctness, therefore, is a function of the chosen delivery specification.

This leads to a central question: who defines correctness, and how can it be verified? Linearizability answers this question by providing a precise and compositional correctness condition that allows programmers to reason about concurrent executions using familiar sequential semantics.

## 2.2 Definition of Linearizability

A *history* $(H)$ is a finite sequence of invocation and response events of operations executed by a set of processes (or threads).

- An *invocation event* has the form $\mathsf{inv}\langle p, op, x\rangle$, meaning process $(p)$ invokes operation $(op)$ with argument $(x)$.
- A *response event* has the form $\mathsf{res}\langle p, op, y\rangle$, meaning process $(p)$ returns from operation $(op)$ with result $(y)$.

A history need not be well-matched: it may contain *pending invocations* (i.e., invocations without corresponding responses).

An operation is *complete* in $(H)$ if its invocation is followed by a matching response in $(H)$.

```
Enq = proc (q : queue, x : item)
   i : int := INC(q.back)     % Allocate a new slot.
   STORE(q.items[i], x)       % Fill it.
end Enq

Deq = proc (q : queue) returns (item)
   while true do
      range : int := READ(q.back) − 1
      for i : int in 1 . . . range do
         x : item := SWAP(q.items[i], null)
         if x ≠ null then return(x) end
      end
   end
end Deq
```

**Figure 1.** Lock-free queue enqueue and dequeue operations.

A history $(H')$ is an *extension* of a history $(H)$ if:

1. $(H)$ is a prefix of $(H')$, and
2. $(H')$ is obtained from $(H)$ by appending zero or more response events corresponding to pending invocations in $(H)$.

Thus, extending a history may complete some pending operations or complete none at all (a zero extension), but it does not introduce any new invocations.

A history $H$ is *linearizable* if it can be extended to a history $H'$ such that:

1. complete$(H')$ is equivalent to a legal sequential history $S$,
2. the real-time order is preserved, i.e., $<_H \subseteq <_S$.

Here, complete$(H')$ denotes the maximal subsequence of $H'$ obtained by removing pending invocations. Linearizability permits nondeterminism: multiple sequential histories may justify the same concurrent execution.

## 2.3 Proposed Solution

To address the verification challenge, we consider the Herlihy-Wing queue (HW hereinafter) as our primary case study. As shown in Figure 1, the HW queue is implemented using an infinite array initialized with **null**. We distinguish between two levels of operation. The **Abstract (ABS)** level comprises the high-level specification methods, `Enq` and `Deq`, which define the external interface of the object. The **Representation (REP)** level consists of the concrete, atomic implementation steps (capitalized in the figure, e.g., `INC`, `STORE`, `SWAP`, `READ`) that compose the ABS methods.

Unlike a sequential specification where a concrete state $r$ maps to a single abstract state $q$ (a function $REP \rightarrow ABS$), a concurrent implementation allows for ambiguity. Due to the overlap of concurrent operations, the concrete memory state may effectively represent multiple valid abstract states simultaneously. To capture this, we define our abstraction

| Event | $A(Lin(H\|_{REP}))$ | $Lin(H\|_{ABS})$ | V |
|---|---|---|---|
| Init | $\{[]\}$ | $\{[]\}$ | ✓ |
| A: ENQ(X) INV | $\{[]\}$ | $\{[], [x]\}$ | ✓ |
| A: INC | $\{[]\}$ | $\{[], [x]\}$ | ✓ |
| A: STORE(X) | $\{[], [x]\}$ | $\{[], [x]\}$ | ✓ |
| A: OK | $\{[x]\}$ | $\{[], [x]\}$ | ✓ |
| B: ENQ(Y) INV | $\{[x]\}$ | $\{[], [x], [y], [x, y], [y, x]\}$ | ✓ |
| B: INC | $\{[x]\}$ | $\{[], [x], [y], [x, y], [y, x]\}$ | ✓ |
| B: STORE(Y) | $\{[x], [x, y]\}$ | $\{[], [x], [y], [x, y], [y, x]\}$ | ✓ |
| B: OK | $\{[x, y]\}$ | $\{[x, y]\}$ | ✓ |

**Figure 2.** Verification trace of concurrent enqueues

function $A(r)$ to map a concrete state to the power set of abstract states:

$$A(r) : REP \rightarrow 2^{ABS}$$

This relaxation allows us to model all possible combinations of valid states that the system could be in, accounting for the non-determinism inherent in concurrency.

In terms of the concrete implementation, if a STORE operation that writes $x$ to the array completes before an INC operation allocates a slot for $y$ , then $x$ is guaranteed to occupy a lower array index than $y$, establishing the order $x <_r y$.

Using this partial order, we define the abstraction function $A(r)$ as the set of all abstract queues $q$ that satisfy two conditions. First, *Content Preservation* requires that the items in the queue match the items in the array ($items(q) = items(r)$). Second, *Order Preservation* requires that the total order of the queue $<_q$ is consistent with the partial order of the array $<_r$; that is, $<_r \subseteq <_q$. Formally:

$$A(r) = \{q \mid items(q) = items(r) \wedge <_r \subseteq <_q\}$$

This definition captures the essence of the HW queue: while items are ordered by index, the concurrent nature of the Enq method (specifically the gap between INC and STORE) means that the "logical" order of enqueues may vary until the values are physically visible in the array.

The ultimate goal is to prove that the implementation is linearizable. We recall that a history $H$ is linearizable if it can be extended to a complete history $H'$ that is equivalent to a legal sequential history $S$. In our set-theoretical model, we denote the set of all such legal sequential histories with the current execution as $Lin(H|_{ABS})$.

$$A(r) \subseteq Lin(H|_{ABS})$$

If this condition holds initially and is preserved by every atomic step (REP transition) of the algorithm, we successfully can say that the program order respects the correctness specification.

This trace, as demonstrated in Figure 2, illustrates the verification process. We observe two concurrent processes: process A enqueuing $x$ and process B enqueuing $y$. Initially, both the concrete array and the set of possible abstract states are empty. When A invokes ENQ and executes INC, the concrete state remains empty (since STORE has not yet occurred), yet the abstraction set expands to include both the empty queue and a queue containing $x$, reflecting the ambiguity of whether A's write is yet visible. The critical moment arrives at A's STORE(X): the value now physically appears in the array, and the abstraction set narrows from $[], [x]$ to reflect this certainty. Before B's STORE(Y), the abstraction set $[x]$ contains only queues with $x$ present; after B's STORE(Y), it expands to $[x], [x, y]$ because the concrete memory now shows both values, but we cannot yet determine the relative order imposed by external observers. The final step, when both operations complete, converges to a single linearizable history: the queue $[x, y]$. Throughout this execution, the invariant $A(Lin(H| * \text{REP})) \subseteq Lin(H| * \text{ABS})$ is maintained, confirming that every state the concrete implementation produces is a subset of the abstract specification.

### 2.4 Why Linearizability is the Standard Correctness Condition

Linearizability has become the standard correctness condition for concurrent objects because it simultaneously preserves real-time semantics, supports modular reasoning. To understand its importance, it is useful to compare it with two closely related conditions: sequential consistency and serializability.

### 2.5 Sequential Consistency

Sequential consistency requires that a concurrent history be equivalent to some legal sequential history, but it does *not* require preservation of real-time order. That is, if operation $e_1$ completes before $e_2$ begins in the actual execution, sequential consistency does not require $e_1$ to precede $e_2$ in the sequential explanation.

Consider a FIFO queue with the following history:

$$\text{ENQ}(x)_A;\ \text{OK}_A;\ \text{ENQ}(y)_B;\ \text{OK}_B;\ \text{DEQ}()_B;\ \text{OK}(y)_B.$$

Here, $x$ is clearly enqueued before $y$, yet $y$ is dequeued first. This history can be made sequentially consistent by reordering the operations as if $\text{ENQ}(y)$ occurred before $\text{ENQ}(x)$. However, this violates the real-time order observed by an external observer. Linearizability forbids such reordering because it requires that the sequential explanation respect the real-time precedence relation.

More importantly, sequential consistency is *not compositional*. Even if each object is sequentially consistent in isolation, the whole system may not be. Herlihy and Wing provide an example with two queue objects $p$ and $q$ where each subhistory is sequentially consistent, but their composition is not. This failure of locality means that objects cannot be verified independently, which severely limits modular design.

### 2.6 Serializability

Serializability requires that concurrent *transactions* be equivalent to some sequential execution. In strict serializability, the sequential order must also respect real-time order.

Serializability is *also* not local. Even if each object's projection of a history is serializable, their combination may not be. Herlihy and Wing give a history involving two objects $p$ and $q$ where each object individually admits a serial explanation, yet no global serial order exists. Thus, serializability does not compose.

Also, serializability is inherently *blocking*. Consider two registers $x$ and $y$ and two transactions:

$$A : \texttt{Read}(x); \texttt{Write}(y) B : \texttt{Read}(y); \texttt{Write}(x)$$

If both reads return 0, neither write can complete without violating serializability. One transaction must abort or wait.

By contrast, linearizability is nonblocking: a pending invocation of a total operation can always be completed without waiting for another operation. This makes it especially appropriate for low-level concurrent data structures and multiprocessor systems.

### 2.7 Compositionality: The Key Distinction

The defining advantage of linearizability is *locality*:

$$H \text{ is linearizable} \iff \forall x.\ H|_x \text{ is linearizable.}$$

Thus objects can be verified independently and then safely composed.

### 2.8 Conclusion

We have established a formal definition of linearizability and the mathematical underpinnings required to prove a piece of code is linearizable. Practical challenges remain, for example, deriving the precise abstraction function $A(r)$ for complex data structure seems to be a non-trivial task and requires some ingenuity. However, these open questions do not diminish the foundational work by Herlihy and Wing. This paper provides the bedrock to successfully talk about *correctness* from an intuition into a provable property.

## 3 Automatically Proving Linearizability [13]

### 3.1 Motivation

We discussed in the previous section that *linearizability* is the standard correctness condition for concurrent objects, yet proving linearizabilty for realistic implementations remains a significant challenge. We explore an alternate way of proving linearizability by identifying *linearization points*: an instant during the operations's execution at which the effect is deemed to occur atomically. While this approach is effective for simple implementations, it becomes increasingly difficult as algorithms grow more sophisticated because of complex control flow or inter-thread interference. In many cases, a linearization point may be conditional, may occur in a different thread, or may depend on future execution. This is particularly common for operations that do not logically modify the shared abstract state, such as unsuccessful lookups or dequeue operations on an empty deque. For such operations, insisting on a concrete linearization point is marked with certain difficulty as it maybe in different places for different outputs.

Consider a lock-free stack with a `pop` operation.

- Scenario A (Success): If the stack is not empty, the linearization point is the successful `CAS` instruction that updates the head pointer.
- Scenario B (Empty): If the stack is empty, the `CAS` is never attempted. The linearization point is instead the `READ` instruction where the thread observed the null head.

At the same time a different category of operations exist, *effectful operations*, different in the sense that they necessarily update the abstract state, and those that are *pure* which do not update the state. Effectful operations often, by established definition of the point where the effect takes place, becomes a natural candidate for linearization points, while pure operations do not. The key insight of the paper is that these two classes of executions should be treated differently, and that linearizability can be established without explicitly committing to a linearization point for every operation.

### 3.2 Proposed Solution

The proposed approach replaces reasoning about linearization points and sequential histories with an instrumentation-based verification strategy. The method augments the abstract specification directly into the concrete implementation through additional state and assertions. Linearizability is then reduced to the problem of showing that certain assertions cannot fail.

The core idea is to distinguish whether an operation has performed an effectful abstract update or whether it has remained pure. For effectful executions, the abstract operation is executed at a candidate instruction that is already present in the implementation, such as a successful compare-and-swap. For pure executions, the method refrains from choosing a specific linearization point; instead, it records which return values are consistent with the abstract specification at some point during the execution.

This distinction is realized by instrumenting each operation with auxiliary variables that track abstract effects. One variable records the result of an effectful abstract execution, if such an execution occurs. In parallel, an array of flags records whether particular return values are admissible for a pure execution according to the specification. Initially, no abstract effect is assumed to have taken place, and no return value is assumed to be valid.

```
int tryDequeue(void) { Node next, head, tail; int
     pval;

   while (true) { head = Q->head; tail = Q->tail
       ; next = head->tl;

      if (Q->head != head) continue;

      if (head == tail) { if (next == NULL)
           return EMPTY;

         CAS(&Q->tail, tail, next); } else {
             pval = next->val; if
       (CAS(&Q->head, head, next)) return pval;
            } } }
```

**Figure 3.** Michael & Scott queue tryDequeue (adapted from Vafeiadis [13]).

As the concrete execution proceeds, candidate effectful linearization points trigger the execution of the abstract specification, and the result of this execution is stored. Independently, whenever the specification permits a pure operation to return a certain value without modifying the abstract state, the corresponding admissibility flag is enabled. At each return point of the concrete method, a single assertion is checked: either an effectful abstract update has occurred and the concrete return value matches the recorded abstract result, or no effectful update has occurred and the returned value is among those permitted by the specification.

### 3.3 Example: Concurrent Queue

The intuition behind the method can be illustrated using a concurrent queue with operations *enqueue* and *dequeue*. Successful enqueue operations are effectful: they necessarily extend the abstract queue. In typical implementations, this effect corresponds to a concrete instruction that links a new node into the data structure. At this instruction, the abstract enqueue operation is executed, and its result is recorded.

In contrast, a dequeue, from Figure 3, operation that returns EMPTY does not modify the abstract queue. Such an execution may span multiple reads and checks, and no single instruction correspond to its logical effect. Rather than assigning it an arbitary linearization point, the method simply checks whether returning EMPTY is consistent with the abstract queue being empty at some point during the execution. If so, the return is deemed admissible.

Determining the linearization point requires distinguishing between varying execution states: whether the queue is empty, whether it is non-empty, and whether the thread is actively "helping" another operation complete. Crucially, this results in a linearization point that is both input-dependent and execution-dependent. For example, if the queue is empty,

```
int tryDequeue(void) { Node next, head, tail; int
     pval;

   // --- instrumentation at entry --- lres =
       UNDEF; can_return[EMPTY] = false;

   while (true) { head = Q->head; tail = Q->tail
       ; next = head->tl;

      if (Q->head != head) continue;

      // PURE CHECKER (symbolic) // if (AQ ==
          empty) can_return[EMPTY] = true;

      if (head == tail) { if (next == NULL) {

            assert( lres == EMPTY || (lres ==
                UNDEF && can_return[EMPTY]) );

            return EMPTY; }

         // --- CAS on tail if (CAS(&Q->tail,
             tail, next)) { assert (lres
           ==UNDEF); lres = spec_tryDequeue();
               // WRONG LP, but still a
           candidate }

      } else { pval = next->val;

         // --- CAS on head if (CAS(&Q->head,
             head, next)) { assert(lres ==
           UNDEF); lres = spec_tryDequeue();
               assert(lres == pval); return
           pval; } } } }
```

**Figure 4.** Instrumented `tryDequeue`

the linearization point is not a state-modifying CAS, but rather the atomic read confirming `next == NULL`. Thus, unlike enqueue, the `tryDequeue` method resists a single static definition, requiring a case-by-case evaluation of the runtime execution path.

### 3.4 Verification Procedure

The verification procedure presented in this work is an automatic technique for proving the linearizability of concurrent data structure implementations. The procedure operates by distinguishing between *effectful* executions (which modify the shared abstract state) and *pure* executions (which do not). The algorithm, formally defined as `PROVELINEARIZABLE`, takes as input the library's constructor, its operations, and their corresponding atomic functional specifications. The procedure is divided into two distinct phases: a *Preparation Phase* that generates auxiliary constructs and a *Checking Phase* that instruments the code for verification.

***Pure Checker Generation.*** For each operation, a "Pure Linearizability Checker" is generated. This checker is derived from the specification by identifying all syntactically pure execution paths—paths that do not assign to global variables. Along these paths, the return statement `return v` is replaced with the assignment `can_return[v] = true`.

***Candidate Linearization Points.*** The function `GETCANDIDATELINPOINTS` analyzes the source code to identify potential effectful linearization points. It unfolds definitions of atomic primitives (like CAS) to expose control flow and selects one state-modifying command along each execution path as a candidate linearization point.

***Instrumentation Variables.*** In the checking phase, each operation $op_i$ is instrumented with two auxiliary variables per thread. The variable `lres` stores the result of the abstract operation if an effectful linearization point occurs; it is initialized to `UNDEF`. The boolean array `can_return`, indexed by possible return values, tracks whether a valid pure linearization point has been observed; all entries are initialized to `false`.

***Code Transformation.*** The source code is modified at three places. At every identified effectful linearization point, code is inserted to execute the abstract specification, and an assertion `assert(lres == UNDEF)` ensures the linearization point executes at most once per path. In the background, the verification tool simulates the execution of the Pure Checker after every atomic command of the current thread and after every step of concurrently executing threads, updating `can_return` continuously based on the state of the abstract object. Finally, at each return point yielding value `res`, the following assertion is inserted:

$$\texttt{assert}\big(lres = res \lor (lres = \texttt{UNDEF} \land can\_return[res])\big)$$

This asserts that either the operation linearized effectively (matching `lres`) or it found a valid pure linearization point (tracked by `can_return`). If the verification tool establishes that all inserted assertions hold on every execution path, the implementation is concluded to be linearizable. Figure 4 presents the instrumented version of the code, where each *CAS* instruction is augmented with assertion-based instrumentation to capture potential effectful linearization points and a pure linearization checker updates `can_return` for the empty-queue case.

### 3.5 Verify and RGSep

After the implementation has been transformed and instrumented with candidate linearisation points and pure-checker assertions, the Verify procedure is invoked as the core proof engine [14]. Verify constructs a *most-general client*: a model of an unbounded number of threads each repeatedly calling the library methods. It then runs an abstract interpretation over this model to establish two properties simultaneously: (i) memory safety is maintained throughout execution, and (ii) none of the instrumented assertions are ever violated. Both obligations are discharged using the RGSep action-inference algorithm [14]. At a very high level a thread's *guarantee* ($G$) is the set of actions it promises to restrict itself to when modifying shared memory, while its *rely* ($R$) is the set of actions it assumes the environment (all other threads) might perform during its execution.

The reason why we need Rely-Guarantee is because standard Hoare logic, $\{P\}\ c\ \{Q\}$, is unsound for concurrent programs: a postcondition established by one thread may be invalidated immediately by an interfering thread before the next instruction executes. RGSep [14] addresses this by confining interference to the shared heap and governing it entirely through actions, so that the verifier only reasons about one thread at a time against a single *global rely* bucket rather than enumerating all possible interleavings.

### 3.6 Worked Example: Inferring Guarantees for the M&S Queue

To make the above concrete, consider the Michael-Scott lock-free queue. The tool infers the thread guarantee $G$ automatically by scanning the code for successful compare-and-swap operations.

***Action $G_1$ :: appending the new node.*** The first successful `CAS(&tail->tl, next, node)` is identified. Inspecting the pre- and post-states of the shared heap yields:

$$G_1 \triangleq \texttt{tail}{\rightarrow}\texttt{tl} \mapsto \texttt{NULL} \rightsquigarrow \texttt{tail}{\rightarrow}\texttt{tl} \mapsto \texttt{node}. \quad (1)$$

ie a thread may advance the `tl` pointer of the current tail node from `NULL` to the freshly allocated `node`.

***Action $G_2$ :: advancing the tail pointer.*** The second successful `CAS(&Q->tail, tail, node)` yields:

$$G_2 \triangleq \texttt{Q}{\rightarrow}\texttt{tail} \mapsto \texttt{tail} \rightsquigarrow \texttt{Q}{\rightarrow}\texttt{tail} \mapsto \texttt{node}. \quad (2)$$

This captures the swing of the global tail pointer from the old tail cell to the new one.

***Composing the guarantee.*** The two discovered mutations are packaged into the thread's full guarantee:

$$G = G_1 \cup G_2. \quad (3)$$

All guarantees are collected into a single *global rely*, $R_{\text{global}} = \bigcup_i G_i$. At each atomic step of the thread under verification, the verifier pauses at the current abstract shared state $S$ and applies $R_{\text{global}}$ to $S$, effectively simulating the environment non-deterministically firing any action from the bucket.

At this point, one might naturally wonder: if we keep applying these operations, won't the possibilities just keep expanding? If every application of the global rely yields a new state that we then have to account for, it seems like the state space would grow infinitely, making it impossible to ever reach a fixed point.

The original paper addresses this exact issue using a Stabilize method, which guarantees that a fixed point is indeed eventually achieved [14]. However, I am omitting the formal details of Stabilize from this report, as its underlying proofs are quite dense, and I admittedly did not understand them well enough to confidently explain them here.

Regardless of the underlying math the high-level takeaway is: because the $R_{\text{global}}$ bucket already contains *every* action any thread could ever perform, verifying a single thread against this one bucket is equivalent to verifying it against an unbounded, arbitrary interleaving of all threads. The state-explosion problem is thereby avoided entirely.

### 3.7 Conclusion

The approach presented in this paper offers a practical and conceptually clean path to automated linearizability proofs. By separating effectful and pure executions and embedding the abstract specification directly into the implementation, it avoids the need for explicit linearization points in cases where they are difficult to identify. Linearizability is reduced to a local safety property, enabling the use of mature verification tools.

The method has been implemented in the Cave verification tool and evaluated on a range of concurrent stacks, queues, and set implementations. The results demonstrate that the approach is expressive enough to handle realistic concurrent data structures while remaining amenable to automation.

## 4 Zoo: A Framework for the Verification of Concurrent OCaml 5 Programs using Separation Logic[1]

### 4.1 Motivation

So far, we have focused on linearizability for data structures in which the linearization point can be identified either within the operation itself or within another method invoked during its execution. We now shift attention to concurrent libraries operating under a language- specific memory model under sequential consistency. The release of OCaml 5 introduced true parallelism via multicore support, enabling a growing ecosystem of concurrent libraries such as Saturn, Eio, and Kcas. While this substantially broadened OCaml's applicability, it also raised new verification challenges, particularly the lack of a practical framework for reasoning about realistic concurrent OCaml programs that rely on features such as algebraic data types and mutable records.

Existing Iris [9] based approaches, including *HeapLang*, are expressive but poorly aligned with OCaml's language features and physical equality semantics. They lack essential language constructs, most notably algebraic data types and mutually recursive functions, and require substantial manual translation from OCaml into the modeling language. This makes realistic verification impractical. Zoo addresses this by introducing ZooLang, a language closely modeling OCaml.

```
(* Intended: atomic field inside a record *) type
     node = { value : int; next :
   node option }

(* Unsafe encoding to simulate atomic field *)
    let cas_next (n : node) oldv newv
  = let atomic_field = (Obj.magic (&n.next) :
      node option Atomic.t) in
  Atomic.compare_and_set atomic_field oldv newv
```

**Figure 5.** Simulating an atomic record field using an unsafe cast.

Figure 5 illustrates an issue and further motivates the need for a new framework. The programmer wishes to treat the record field `next` as atomic, but since the type system does not permit atomic annotations on fields, the code casts the field's memory to `Atomic.t` using `Obj.magic`. At this point, the type system's guarantees are circumvented: the field is declared nonatomic in the type definition.

The consequence is that the semantic classification of the location (as atomic or nonatomic) no longer aligns with the static program structure. This breaks the abstraction boundary and complicates formal reasoning, since the memory model and verification logic must now account for behavior that is invisible at the type level.

*Heaplang* does not even know what a record is. To use *HeapLang*, we would have to chop up the record and pretend it is made of basic pairs. If we do that, our proof does not match the real OCaml memory layout anymore. The proof becomes totally disconnected from the actual source code and we risk not checking the real program.

The paper is motivated by these limitations. Its goal is to develop a practical verification framework for real-world concurrent OCaml programs, while refining the language and its semantics to better support safe and verifiable programs. Importantly, the framework continues to assume *sequential consistency* (SC) as the underlying memory model. That is, all atomic operations are reasoned about as if they occur in a single global total order that respects program order.

### 4.2 Proposed Solution

The authors propose Zoo, a comprehensive framework for verifying concurrent OCaml 5 programs using Iris.

***ZooLang.*** At the core of Zoo is ZooLang, a language designed to faithfully model a substantial fragment of OCaml. ZooLang supports algebraic data types, mutable and immutable records, references, atomic operations, mutual recursion, and concurrency primitives. It is deeply embedded in

Rocq and comes with a formally defined operational semantics and a corresponding Iris-based program logic. The framework includes a tool, ocaml2zoo, which translates OCaml source programs into ZooLang code embedded in Rocq. ZooLang is syntactically very close to OCaml, ensuring that verification artifacts remain aligned with real implementations.

***Specifications and Proofs.*** Once translated to ZooLang, users can write specifications and prove them in Iris. For example, the specification of `stack_push` is:

```
Lemma stack_push_spec t ℓ v :
    <<< stack_inv t ℓ
        | ∀∀ vs,  stack_model t vs >>>
        stack_push t v @ ↑ ℓ
    <<< stack_model t (v :: vs)
        | RET (); True >>> .
```

As in Hoare logic[8], the specification consists of pre- and postconditions, each split into a private part (the stack invariant stack_inv t) and an atomic part (the abstract stack state stack_model). The atomic conditions capture the linearization point, stating that during execution the stack's abstract state is atomically updated from `vs` to `v :: vs`. The precise semantics of this atomic triple and its proof obligations are unpacked in detail in the subsection 4.4 *Worked Example: Bounded MPMC Stack*.

***Physical Equality: A Precise Semantics.*** An important aspect of the paper concerns the semantics of physical equality and `compare_and_set`. While this is not presented as a primary technical contribution of the work, it emerged as a critical issue during the verification effort. Physical equality is indispensable for lock-free algorithms, since CAS relies on pointer identity rather than structural equality. However, in OCaml, physical equality (==) is intentionally underspecified and can be affected by compiler optimizations such as sharing and unsharing. During formalization, the authors observed that naively equating physical equality with structural equality is unsound. In particular, compiler induced unsharing can break assumptions required for CAS correctness, making concurrent reasoning unstable.

To address this, the paper models physical equality as nondeterministic but subject to carefully specified guarantees. Moreover, the authors introduce generative constructors as a disciplined mechanism to control unsharing. These constructors ensure that freshly allocated values have stable identities, so that physical equality once established remains meaningful for reasoning. This insight—arising directly from the verification process—clarifies the semantic requirements necessary for sound CAS-based reasoning in OCaml's concurrency model.

Physical equality in OCaml is not fully determined by the abstract value semantics. For example:

```
Some 0 == Some 0
```

| Source Program | Heap Layout |
|---|---|
| `(* After compiler unsharing *)`<br><br>`let x = Some 0 in`<br>`let y = Some 0 in x == y`<br>`(*false *)` | After unsharing:<br>x —-> [Some \| 0] (B1)<br>y —-> [Some \| 0] (B2) |

**Figure 6.** Unsharing of immutable blocks in OCaml. The abstract value `Some 0` remains the same, but physical identity changes due to compiler duplication.

may return `true` or `false` depending on whether the two blocks are represented identically in memory. Zoo formalizes this by distinguishing:

- $\hat{v}_1 == \hat{v}_2$: the values *must* be physically equal,
- $\hat{v}_1 \approx \hat{v}_2$: the values *may* be physically equal.

This distinction allows the semantics to avoid baking in false assumptions about representation sharing, acknowledging that the compiler may make implementation-specific choices about block allocation and layout.

As shown in Figure 6, the original program binds `y` to `x`, so both variables reference the same heap block. Since OCaml's `==` operator checks physical equality (pointer identity), `x == y` evaluates to `true`.

After compiler unsharing, the expression `Some 0` is duplicated. Although the abstract values are structurally equal, they now occupy distinct heap blocks. Consequently, `x == y` evaluates to `false`.

This phenomenon is semantically invisible under structural equality but observable under physical equality. In concurrent code, where algorithms sometimes rely on pointer identity for synchronization, such unsharing can introduce correctness bugs.

### 4.3 CAS Vulnerability to Unsharing

Compare-and-swap (CAS) operations rely on physical equality:

```
Atomic.compare_and_set loc expected new
```

succeeds only if the current value at `loc` is physically identical to `expected`.

If the compiler unshares `expected`, CAS behavior can change unexpectedly. Initially, when `expected` and the value at `loc` reference the same block, CAS succeeds. After unsharing, `expected` becomes a distinct copy, so CAS fails despite abstract equality. Thus, program behavior may vary depending on compiler optimizations, making concurrent reasoning fragile.

Zoo introduces *generative immutable blocks*, marked with the `[@generative]` annotation.

For generative blocks, physical equality is tightly specified:

$$\hat{v}_1 == \hat{v}_2 \iff bid_1 = bid_2$$

This makes them behave like mutable references in terms of identity, even though their contents are immutable.

The key invariant now is *the compiler must not unshare generative blocks.*

When a constructor is marked `[@generative]` the compiler preserves the block's identity across the entire computation. Physical equality becomes deterministic with respect to that block and we enforce two generative blocks are physically equal iff they have the same `bid`.

Without `@generative`, physical equality is under-specified: two abstractly equal values may or may not be physically equal, meaning CAS correctness depends on compiler behavior rather than program logic. The identity becomes explicit through the `[@generative]` annotation, physical equality is no longer under-specified for generative blocks, and CAS reasoning becomes stable and provable. Zoo demonstrates this with the Rcfd (recursive file descriptor) example where verification fails when the file descriptor state is non-generative, but succeeds after marking it `[@generative]`, because CAS operations on the state can now be relied upon. The intuition is : *generative* constructors tell the compiler this value has identity ; do not clone it. This restores the invariant that if CAS fails, it is because the concurrent state truly changed, not because the compiler duplicated a value.

### 4.4 Worked Example :: Bounded MPMC Stack

I will now extract the central proof ingredients from the Iris development of `mpmc_bstack` as an example. The full definitions are shown in Figure 7. I will highlight the lemmas that make the verification go through and explain their logical role.

***Injectivity.*** The physical representation of the stack is an encoded value `lst_to_val (length vs) vs`. When CAS succeeds, we obtain physical equality between the stored value and the one we read. Injectivity then lifts this physical equality to Rocq-level equality of the abstract lists. In other words, this lemma bridges the gap between operational equality and mathematical equality.

When the compare-and-swap works, we find out that the `front` of the stack is exactly `lst_to_val (length vs) vs`, and the value in memory is physically equal to what we read before. Operationally, this gives us the neat equation

$$\texttt{lst\_to\_val (length } vs_1\texttt{) } vs_1 \\ = \texttt{lst\_to\_val (length } vs_2\texttt{) } vs_2 \tag{4}$$

. The proof obligation, thus is that, we must conclude that the abstract lists themselves are equal just from the equality of these encoded values. We need a strict rule that says $f(vs_1) = f(vs_2) \Rightarrow vs_1 = vs_2$. (which is injectivity).

***Twins algebra and agreement.*** The ghost state uses the Twins resource algebra to split the abstract state into two halves. For example, the client holds $\text{model}_1\ \gamma\ vs_1$. This has three specific parts: $\text{model}_1$ is the client's resource token, $\gamma$ is a unique ghost name (like a serial number) linking it to the invariant's $\text{model}_2$ token, and $vs_1$ is the pure Rocq list representing the abstract state. Holding this means the client owns the token for stack $\gamma$ and asserts the state is currently $vs_1$.

When we execute an operation, the `model_agree` lemma brings the halves together. Because the $\gamma$ identifiers match, it mathematically proves the lists are identical, giving us the fact $vs_1 = vs_2$.

Iris is built to manage spatial memory resources. The separating conjunction ($P * Q$) strictly requires both $P$ and $Q$ to be valid resources. If we attempt to connect a memory proposition $P$ directly with a pure mathematical fact $Q$ (such as $vs_1 = vs_2$), the system will throw a type error. To bridge this gap, we wrap the pure fact inside a special modality, written as $\ulcorner vs_1 = vs_2 \urcorner$. This embedding translates the mathematical truth into an Iris resource that formally consumes exactly zero memory. This mechanism enables us to safely construct formulas like $\text{model}_1\ \gamma\ vs_1 * \ulcorner vs_1 = vs_2 \urcorner$, placing the stateful resource and the mathematical fact legally side by side.

```
(* Injectivity of the encoding *) Lemma
    lst_to_val_inj' vs1 vs2 : lst_to_val
  (length vs1) vs1  lst_to_val (length vs2) vs2
      -> vs1 = vs2.

(* Ghost state agreement *) Lemma model_agree
    vs1 vs2 : model₁  vs1 -∗model₂
   vs2 -∗vs1 = vs2.

(* Ghost state update *) Lemma model_update { vs1
     vs2} vs : model₁  vs1 -∗
  model₂  vs2 ==∗model₁  vs ∗model₂  vs.

(* Capacity validity *) Lemma
    mpmc_bstack_model_valid t  cap vs :
  mpmc_bstack_inv t  cap -∗mpmc_bstack_model t vs
       -∗length vs  cap.

(* Main push specification *) Lemma
    mpmc_bstack_push_spec t  cap v : ⟪⟪⟪
  mpmc_bstack_inv t  cap | ∀∀vs,
      mpmc_bstack_model t vs ⟫⟫⟫mpmc_bstack_push t
  v @ ^ ⟪⟪⟪∃∃b, b = bool_decide (length vs < cap)
      ∗mpmc_bstack_model t (if
  b then v :: vs else vs) | RET #b; True ⟫⟫⟫.

(* Pop specification *) Lemma
    mpmc_bstack_pop_spec t  cap : ⟪⟪⟪
    mpmc_bstack_inv
  t  cap | ∀∀vs, mpmc_bstack_model t vs ⟫⟫⟫
      mpmc_bstack_pop t @ ^ ⟪⟪⟪
  mpmc_bstack_model t (tail vs) | RET head vs;
      True ⟫⟫⟫.
```

**Figure 7.** Core lemmas used in verifying `mpmc_bstack`.

***Capacity reasoning.*** The lemma `mpmc_bstack_model_valid` extracts the key safety property of the bounded stack: any abstract list described by `mpmc_bstack_model` must satisfy `length vs <= cap`. This fact is used in the push proof to justify the branch where insertion is disallowed when the stack is full. Thus, logical capacity constraints are enforced at the specification level.

***Push and pop specifications.*** An atomic triple is written as follows:

$$\langle\!\langle\ \forall\forall vs,\ P\ vs\ \rangle\!\rangle\ e\ \langle\!\langle\ \exists\exists r,\ Q\ vs\ r\ \rangle\!\rangle \tag{5}$$

The double angle brackets $\langle\!\langle\ \cdot\ \rangle\!\rangle$ distinguish this from a standard Hoare triple $\{P\}\ e\ \{Q\}$. The precondition $P\ vs$ and postcondition $Q\ vs\ r$ refer to the *shared abstract state* of the data structure. Standard Hoare logic assumes a sequential execution: it relies on the guarantee that no other thread will modify the state between the pre-condition and the post-condition. We need a different notation for concurrent systems because this assumption fails; we must always account for the possibility of competing threads invalidating the pre and post conditions.

Lemma `mpmc_bstack_push_spec` in Figure 7 follows a similar syntax structure.

The above atomic triple is *syntactic sugar*; its full meaning is given by the following Weakest Precondition formula:

$$\forall\Phi.\ \Big(\forall vs,\ P\ vs \mathrel{-\!\!*} \big(\forall r,\ Q\ vs\ r \mathrel{-\!\!*} \Phi\ r\big)\Big) \mathrel{-\!\!*} WP\ e\ \{\{\Phi\}\} \tag{6}$$

Before I dive into the what the verification conditions are, lets unpack the above equation 6. expression. Separation logic extends classical logic with a *spatial* connective called the *magic wand* (or *separating implication*), written $A \mathrel{-\!\!*} B$. The proposition $A \mathrel{-\!\!*} B$ denotes a *resource exchange contract*: if you provide the resource $A$, it is consumed and the resource $B$ is returned in its place. Ownership of $A$ is *transferred away* from the caller; ownership of $B$ is *transferred back*. This transfer semantics is essential in concurrent settings because it prevents two threads from simultaneously claiming ownership of the same heap cell.

A standard Hoare triple $\{P\}\ e\ \{Q\}$ is encoded in Iris as $P \mathrel{-\!\!*} WP\ e\ \{\{Q\}\}$, encoding a static contract in which the precondition $P$ must hold *before* $e$ begins and $Q$ holds *after* it terminates. However, this is insufficient for concurrent data structures. Because other threads may modify the shared state between any two instructions of $e$. The specification must instead be parameterised over *any possible caller goal*, which motivates the leading universal quantifier $\forall\Phi$ in the desugared form (6). The Weakest Precondition $WP\ e\ \{\{\Phi\}\}$ asserts that expression $e$ executes safely and that, upon returning a value $r$, the proposition $\Phi\ r$ holds, where $\Phi : val \to iProp$ encodes whatever the calling thread requires of the returned value.

The push specification (Figure 7) is stated as the following atomic triple:

$$\begin{aligned}&\langle\!\langle\ \forall vs,\ mpmc\_bstack\_model\ t\ vs\ \rangle\!\rangle\ mpmc\_bstack\_push\ t\ v\ @\ \iota\\ &\quad\langle\!\langle\ \exists b,\ \ulcorner b = bool\_decide(|vs| < cap)\urcorner\\ &\qquad * \ mpmc\_bstack\_model\ t\\ &\qquad (\text{if } b \text{ then } v :: vs \text{ else } vs)\ \rangle\!\rangle\end{aligned} \tag{7}$$

The four components of this specification are as follows.

| Component | Formula | Role |
|---|---|---|
| Private pre | $mpmc\_bstack\_inv\ t\ \iota\ cap$ | Thread must own this before the call. |
| Public pre | $\forall vs,\ mpmc\_bstack\_model\ t\ vs$ | Shared state at the linearization point. |
| Public post | $\exists b,\ \ulcorner\cdots\urcorner * mpmc\_bstack\_model\ t\ \cdots$ | Updated state after the lineraization point closes. |
| Private po | $RET\ \#b;\ \top$ | Boolean $b$; no extra resources. |

Expanding the syntactic sugar of (7) yields the following raw Weakest Precondition judgment, which constitutes the actual proof obligation discharged in Iris:

$$\begin{aligned}&mpmc\_bstack\_inv\ t\ \iota\ cap \mathrel{-\!\!*} \forall\Phi.\\ &\quad\Big(\forall vs,\ mpmc\_bstack\_model\ t\ vs \mathrel{-\!\!*}\\ &\qquad\big(\forall b,\ \ulcorner b = bool\_decide(|vs| < cap)\urcorner\\ &\qquad\quad * \ mpmc\_bstack\_model\ t\ (\text{if } b \text{ then } v :: vs \text{ else } vs)\\ &\qquad\quad \mathrel{-\!\!*} \Phi\ \#b\big)\Big)\\ &\quad \mathrel{-\!\!*} WP\ mpmc\_bstack\_push\ t\ v\ @\ \iota\ \{\{\Phi\}\}\end{aligned} \tag{8}$$

When a developer claims that their MPMC stack implementation is correct, the formal claim is precisely that the implementation *satisfies specification* (8): given that the current thread owns the invariant token $mpmc\_bstack\_inv\ t\ \iota\ cap$, the code of *mpmc_bstack_push* will safely terminate and deliver a result satisfying $\Phi$. Before the linearization point is reached, the client has no direct access to the abstract stack state $vs$; the state is locked inside the invariant. Only at the *single atomic step* corresponding to the linearization point does the proof open the invariant, observe $vs$, perform the update, and re-close the invariant, ensuring that the client's view of the state is *instantaneous and consistent*.

To make this concrete, consider a bounded stack with capacity $cap = 1$. Thread A begins a push of element $e_1$, reads the current length, and observes that the stack is empty ($|vs_1| = 0 < 1$), concluding that insertion is permitted. Before Thread A proceeds to the write step, Thread B completes a full push of element $e_2$, bringing the stack to capacity ($|vs_2| = 1 = cap$). Thread A then resumes and attempts to write $e_1$, believing (incorrectly) that the stack is still empty.

The resulting state contains two elements in a stack whose capacity is one, reductio ad absurdum.

The atomic triple specification forces any proof of correctness to mirror the true atomicity of the operation. Thread A's proof must open the invariant *mpmc_bstack_inv t ι cap* to access the abstract model token *mpmc_bstack_model t vs*, extracting the fact that $|vs_1| < cap$. However, because the *read* instruction is a single atomic machine step, the proof must immediately close the invariant and return the model token to the shared pool. While Thread A has returned the invariant token, Thread B is free to open the invariant, push $e_2$, and close it, advancing the abstract state from $vs_1$ to $vs_2 = e_2 :: vs_1$. When Thread A subsequently attempts the write and re-opens the invariant, Iris enforces a crucial discipline: because the invariant was closed and reopened in the interim, the proof is given a *fresh, universally quantified* abstract state with no information relating $vs_2$ to $vs_1$. To justify the write, the proof must discharge $|vs_2| < cap$, but the only capacity fact in scope is the now-stale $|vs_1| < cap$. There is no proof term that bridges the two abstract states, and the Iris proof checker is left with the stuck goal

$$|vs_1| < cap \quad \not\vdash \quad |vs_2| < cap.$$

***Conclusion.*** This paper introduces Zoo, a practical framework for verifying concurrent OCaml 5 programs, bridging real-world code and mechanized verification through ZooLang. It also contributes language improvements such as atomic record fields and a precise semantics for physical equality. Using Zoo, the authors verified a subset of the OCaml standard library, components of the Eio library, and a large portion of the Saturn lock-free data structure library, including stacks, queues, bags, and a work-stealing deque. These results demonstrate the practicality and scalability of the framework. The project remains actively developed and last semester, I made couple of minor contributions to the codebase.[1][2]

# 5 Efficient Multi-word Compare and Swap[3]

## 5.1 Motivation

At this stage, the verification landscape includes several automated linearizability provers and checkers, supporting both stateless and stateful exploration of concurrent executions.

We now turn to a different flavored data structure: multi-word compare-and-swap (MCAS), as verified in Zoo [1]. Unlike simpler concurrent structures, MCAS exhibits non-local completion: an operation may be completed by a thread other than the one that invoked it, a phenomenon also observed in the automatic linearizability work of Vafeiadis [13]. This helping behavior complicates reasoning, as the linearization point may lie outside the extent of the calling thread.

[1] https://github.com/clef-men/zoo/pull/1
[2] https://github.com/clef-men/zoo/pull/2

Our goal is therefore to examine the complexities involved in defining and verifying a data structure such as MCAS, and to understand how it is engineered for multicore systems. MCAS is specifically designed to exploit fine-grained concurrency rather than relying on coarse-grained mutual exclusion. Its correctness relies on descriptor-based indirection, cooperative helping, and careful synchronization protocols. For this reason, MCAS becomes a compelling and a necessary case study.

## 5.2 Proposed Solution

MCAS generalizes CAS to multiple memory locations. An MCAS operation over $k$ locations atomically performs the following logical action:

> If all locations $x_1, \ldots, x_k$ contain values $o_1, \ldots, o_k$, then update them to $n_1, \ldots, n_k$; otherwise, leave all locations unchanged.

**5.2.1 Classical Descriptor-Based MCAS.** Traditional lock-free MCAS implementations use a descriptor to coordinate the update across multiple locations [5]. The protocol proceeds in three phases.

***Example.*** Consider three memory locations:

$$x = 1, \quad y = 2, \quad z = 3$$

We wish to atomically perform:

$$(x, y, z) : (1, 2, 3) \rightarrow (4, 5, 6)$$

Here, $k = 3$.

***Phase 1: Freeze (Installation) Phase :: $k$ CAS operations.*** Each target location is updated using CAS from its expected value to a pointer to a shared descriptor $D$. This CAS both verifies that the location holds the expected value and installs the descriptor pointer into that word.

$$\begin{gathered}\texttt{CAS}(x, 1, \&D)\\ \texttt{CAS}(y, 2, \&D)\\ \texttt{CAS}(z, 3, \&D)\end{gathered}$$

If any CAS fails, the operation aborts.

***Phase 2: Decision Phase :: 1 CAS operation.*** The descriptor's status is atomically updated to record the outcome:

$$\texttt{CAS}(D.\text{status}, \text{ACTIVE}, \text{SUCCESS})$$

This CAS constitutes the linearization point of the MCAS operation.

***Phase 3: Cleanup Phase :: $k$ CAS operations.*** Each frozen location is updated based on the descriptor's final state:

$$\begin{gathered}\texttt{CAS}(x, \&D, 4)\\ \texttt{CAS}(y, \&D, 5)\\ \texttt{CAS}(z, \&D, 6)\end{gathered}$$

The number of CAS operations are:

$$k \text{ (freeze)} + 1 \text{ (decision)} + k \text{ (cleanup)} = 2k + 1 \qquad (9)$$

For the example above, the result is 7 CAS operations.

### 5.3 Efficient MCAS: $(k + 1)$ CAS

This paper observes that the cleanup phase is conceptually unnecessary: once the global decision has been made, the final value of each location is already fixed at the logical level.

The central insight of this paper's protocol is the following:

*Rather than restoring or overwriting memory locations after the decision, encode the final outcome in the descriptor and interpret each location's value through the descriptor.*

Consequently, a memory location that still references a descriptor is treated as logically containing either the old value or the new value, depending solely on the descriptor's finalized state.

We illustrate the protocol using the same running example:

$$x = 1, \quad y = 2, \quad z = 3$$

The intended atomic update is:

$$(x, y, z) : (1, 2, 3) \rightarrow (4, 5, 6)$$

***Phase 1: Freeze Phase :: $k$ CAS Operations.*** Each target location is atomically updated from its expected value to a pointer to a shared descriptor $D$:

$$\texttt{CAS}(x, 1, \&D)$$
$$\texttt{CAS}(y, 2, \&D)$$
$$\texttt{CAS}(z, 3, \&D)$$

As in classical MCAS, each CAS simultaneously checks the expected value and marks the location as participating in the ongoing MCAS.

***Phase 2: Decision Phase :: 1 CAS Operation.*** The outcome of the operation is finalized by atomically updating the descriptor's status:

$$\texttt{CAS}(D.\text{status}, \text{ACTIVE}, \text{SUCCESS})$$

This operation serves as the linearization point for the MCAS.

***Elimination of the Cleanup Phase.*** In contrast to classical MCAS, the efficient MCAS [3] performs no explicit cleanup CAS's.

- After the decision phase, memory locations may continue to physically reference the descriptor.
- When a thread reads such a location, it consults the descriptor:
  - If the descriptor state is `SUCCESS`, the location is interpreted as holding the new value.
  - If the state is `FAILED`, the location is interpreted as holding the original value.
- As a result, the memory is logically updated without requiring immediate physical rewrites.

The total number of CAS operations is therefore:

$$k \text{ (freeze)} + 1 \text{ (decision)} = k + 1 \tag{10}$$

```
1   readInternal(void *addr, MCASDescriptor
        *self) { retry_read: val = *addr;
2
3   if (!isDescriptor(val)) { return <val,
        val>; }
4
5   MCASDescriptor *parent = val->parent;
6
7   if (parent != self && parent->status ==
        ACTIVE) { MCAS(parent); goto
      retry_read; } else { return
8         parent->status == SUCCESSFUL ?
          <val, val->new>
9       : <val, val->old>; } }
```

**Figure 8.** The `readInternal` auxiliary function: handling descriptors and helping.

For the three-location example, this reduces the cost from 7 CAS operations to 4 CAS operations.

### 5.4 Implementation Overview

The code in Figures 8 and 9 presents a simplified, essential representation of the Guerraoui et al. MCAS algorithm, omitting memory reclamation details and epoch tracking for clarity. The core mechanisms of helping, locking, and non-local linearization are fully preserved in this minimal form.

***The* `readInternal` *Function.*** The `readInternal` function (Figure 8) begins by directly reading the memory location (Line 4). If the location does not contain a descriptor pointer, it immediately returns the value as-is (Line 6). This happy path handles the common case where no concurrent MCAS operation has touched the location. However, if a descriptor pointer is discovered (Line 9), the thread obtains the parent descriptor and must determine whether that operation is still active.

The helping mechanism kicks in at Lines 11-14. When a thread encounters an ACTIVE descriptor whose parent is not its own operation, it recursively calls `MCAS(parent)` to drive that operation to completion before proceeding (Lines 12-13). This ensures that no ACTIVE descriptor can remain indefinitely; any thread observing it will complete it. The self-check (`parent != self`) prevents an operation from recursively helping itself. After helping completes, the thread retries the read to observe the finalized state (Line 14). If the descriptor is already finalized (either SUCCESSFUL or FAILED), the thread interprets its result: for successful operations, it returns the new value; for failed operations, it returns the old value (Lines 16-19). This design ensures that reads never observe partial states ; they see either the state before the operation's linearization point or after it.

```
1     read(void *address) { <content, value>
          = readInternal(address, NULL);
          return
2       value; }
3
4  MCAS(MCASDescriptor *desc) { success = true;
5
6    for wordDesc in desc->words { retry_word:
         <content, value> =
7      readInternal(wordDesc.address, desc);
8
9        if (content == &wordDesc) { continue;
             }
10
11       if (value != wordDesc.old) { success
             = false; break; }
12
13       if (desc->status != ACTIVE) { break; }
14
15       if (!CAS(wordDesc.address, content,
             &wordDesc)) { goto retry_word; } }
16
17   if (CAS(&desc.status, ACTIVE, success ?
         SUCCESSFUL : FAILED)) {
18     retireForCleanup(desc); }
19
20   returnValue = (desc.status ==
         SUCCESSFUL); return returnValue; }
```

**Figure 9.** The read and MCAS operations: two-phase protocol (locking and finalization).

***The* `read` *Operation.*** The `read` operation (Figure 9, Lines 1-4) is a simple wrapper around `readInternal` with `self` set to NULL, indicating that this is not part of an MCAS operation.

***The MCAS Operation: Two-Phase Structure.*** The MCAS operation exhibits a clear two-phase structure: locking and finalization.

In the *locking phase* (Figure 9, Lines 6-15), the operation iterates over each target word in sorted order. For each word, it first calls `readInternal` (Line 7) to obtain the current value while handling any helping obligations. If the word already points to the current descriptor (Line 9), the thread moves to the next word indicating that another thread (or a prior attempt in the current operation) has already locked this location. If the current value does not match the expected old value (Lines 11), the operation must fail, so the thread breaks the loop and proceeds to finalization. If the operation status has changed (Line 13), the loop exits to prevent re-acquiring locations. Finally, the thread attempts a CAS to install a pointer to the descriptor at the target location (Lines 15). If the CAS fails, the thread retries (back to `retry_word`), as the failure may indicate that another thread has concurrently helped this same operation to lock the same word.

Once a location is locked by a descriptor, it remains locked. When all locations are either already locked or successfully locked, the operation enters the *finalization phase* (Lines 17-18). Here, a single CAS atomically changes the descriptor status from ACTIVE to either SUCCESSFUL (if all locking succeeded) or FAILED (if any value mismatch was detected). This status CAS is the linearization point (Line 17): it marks the instant at which the multi-word update becomes logically atomic. The status field determines whether subsequent reads observe the new or old values. Only one thread can successfully perform this CAS (due to the atomic nature of CAS), ensuring that the operation is finalized exactly once.

***Non-Local Linearization Points.*** The non-locality of the linearization point emerges clearly from this structure. Readers determine the logical value of a location by examining the descriptor status, which may be modified on a completely different memory location than the data words themselves. When a read observes a locked location (one pointing to a descriptor), it must wait for that descriptor's status to be finalized, at which point it learns whether the associated update succeeded. Thus, the effect of a k-word MCAS is determined by a single CAS on the descriptor's status field, not by k separate updates to data. In the common uncontended case, an MCAS requires only $k + 1$ CASes (one per data word to lock, plus one for the status).

### 5.5 Conclusion

MCAS is therefore included not as an isolated or exceptional case, but as a deliberate benchmark for expressiveness. Its correctness fundamentally depends on non-local linearization and helping-driven interference.

## 6 Cosmo: A Concurrent Separation Logic for Multicore OCaml [10]

### 6.1 Motivation

So far, we have established a framework for verifying concurrent programs under sequential consistency (SC). Under SC, the execution of a concurrent program can be understood as an interleaving of thread-local actions over a single, centralized shared memory, which makes reasoning about correctness comparatively straightforward.

However, real-world programming languages, such as OCaml, do not operate under sequential consistency. Instead, they adopt relaxed (or weak) memory models, where program executions are not necessarily representable as simple interleavings of thread actions. As a result, correctness reasoning must account for the semantics of the underlying memory model before addressing higher-level concurrent behavior.

To reason about concurrent programs in such languages, one must first reason at a low level, directly in terms of the memory model and its allowed reorderings and visibility constraints. The terms weak memory model and relaxed memory model are used interchangeably, both referring to the fact that program execution is no longer constrained to behave as if all threads interact with a single, sequentially consistent memory. Instead, memory effects may be reordered or observed differently by different threads.

Once a framework for low-level reasoning is in place, it becomes possible to reason modularly about libraries of concurrent data structures, and subsequently about the high-level programs that rely on them.

Fortunately, the Multicore OCaml memory model is significantly simpler to reason about than other managed memory models, such as the Java Memory Model (JMM). A compelling example that corroborates this is the absence of *word tearing*. In Java, if 64-bit values (like `long` or `double`) are not declared `volatile`, the specification allows a thread to observe a torn read ; seeing the top 32 bits of an old value combined with the bottom 32 bits of a new value. In contrast, OCaml values are almost universally word-sized (represented as pointers or tagged integers). Even larger values like `float` are typically boxed and handled as atomic pointer updates, ensuring that word tearing is generally unobservable in OCaml. It distinguishes between two kinds of memory locations: atomic and nonatomic. Atomic locations come with well-defined synchronization guarantees, whereas nonatomic locations must be accessed in a data-race-free manner to ensure sequential consistency.

Reasoning about atomic locations follows a similar spirit to the approach discussed in earlier work on linearizability-based verification [13]. In particular, when an operation performs a stateful change, it becomes possible to identify a clear correctness condition tied to the update. In contrast, stateless operations require the logic to retain and compose all relevant information to justify correctness.

Cosmo builds precisely such a framework: one that supports low-level reasoning about relaxed memory behaviors while enabling compositional, high-level reasoning about concurrent programs and data structures.

$$
\begin{aligned}
a &\in \mathsf{Loc}_{\mathsf{NA}} \\
A &\in \mathsf{Loc}_{\mathsf{AT}} \\
t \in \mathsf{Time} &\triangleq \mathbb{Q} \cap [0,\infty) \\
h \in \mathsf{Hist} &\triangleq \mathsf{Time} \xrightarrow{fin} \mathsf{Val} \\
V, W, G \in \mathsf{View} &\triangleq \mathsf{Loc}_{\mathsf{NA}} \xrightarrow{set} \mathsf{Time} \\
\sigma \in \mathsf{Store} &\triangleq (\mathsf{Loc}_{\mathsf{NA}} \xrightarrow{fin} \mathsf{Hist}) \\
&\quad \times (\mathsf{Loc}_{\mathsf{AT}} \xrightarrow{fin} (\mathsf{Val} \times \mathsf{View}))
\end{aligned}
$$

**Figure 10.** Semantic objects: locations, time stamps, histories, views, and stores.

$$
\textsc{Mem-AT-Alloc}\ \frac{A \notin \mathrm{dom}\,\sigma}{\sigma; W \xrightarrow{\mathrm{alloc}(A,v)} \sigma[A \mapsto (v, W)]; W}
$$

$$
\textsc{Mem-NA-Read}\ \frac{h = \sigma(a) \quad t \in \mathrm{dom}\, h \quad W(a) \le t \quad v = h(t)}{\sigma; W \xrightarrow{\mathrm{rd}(a,v)} \sigma; W}
$$

$$
\textsc{Mem-AT-Read}\ \frac{\sigma(A) = (v, V)}{\sigma; W \xrightarrow{\mathrm{rd}(A,v)} \sigma; W \sqcup V}
$$

$$
\textsc{Mem-NA-Write}\ \frac{h = \sigma(a) \quad t \notin \mathrm{dom}\, h \quad W(a) < t \quad h' = h[t \mapsto v]}{\sigma; W \xrightarrow{\mathrm{wr}(a,v)} \sigma[a \mapsto h']; W[a \mapsto t]}
$$

$$
\textsc{Mem-AT-Write}\ \frac{\sigma(A) = (v, V) \quad V' = W' = W \sqcup V}{\sigma; W \xrightarrow{\mathrm{wr}(A,v')} \sigma[A \mapsto (v', V')]; W'}
$$

$$
\textsc{Mem-AT-Read-Write}\ \frac{\sigma(A) = (v, V) \quad V' = W' = W \sqcup V}{\sigma; W \xrightarrow{\mathrm{rdwr}(A,v,v')} \sigma[A \mapsto (v', V')]; W'}
$$

**Figure 11.** Operational Semantics

### 6.2 Proposed Solution

Figure 10 presents the core semantic objects that explain Cosmo's formalization of the Multicore OCaml memory model. The diagram introduces a small collection of structured entities— locations, time stamps, histories, views, and stores.

Memory locations are partitioned into nonatomic locations $a \in \mathrm{Loc}_{na}$ and atomic locations $A \in \mathrm{Loc}_{at}$. Time stamps $t \in \mathrm{Time}$ are drawn from the nonnegative rationals and are used to order write events at nonatomic locations. A history $h \in \mathrm{Hist}$ is a finite map from time stamps to values; it records, for a single nonatomic location, the sequence of writes that have occurred, each represented as a timestamp–value pair.

Views $V \in \mathrm{View}$ are total mappings from nonatomic locations to time stamps (almost everywhere zero). Intuitively, a view represents *visibility* for each location; it records the most recent write that a thread is aware of. Each thread carries its own current view $W$, which constrains which writes it is allowed to observe and determines how its knowledge evolves over time.

Finally, the store $\sigma \in \mathrm{Store}$ aggregates these components. It consists of a nonatomic store mapping each nonatomic

location to its history, and an atomic store mapping each atomic location to a pair consisting of a value and a view. In this way, the figure showcases how writes are created (by extending histories), how they are ordered (via time stamps), and how visibility is transferred (through the views associated with atomic operations). Put plainly, the semantics explains how Cosmo keeps track of who can see what is in the memory, and which operations only read that information versus which ones actively share it with others.

**BaseCosmo**

The Figure 11 describes the operational semantics of Cosmo describe how a program executes at the machine level: an expression (e) reduces to a new expression (e'), possibly interacting with the memory subsystem through a memory event and possibly spawning new threads. These semantics precisely track how memory locations, histories, timestamps, and views evolve during execution. However, while such a semantics tells us *what can happen*, it does not by itself tell us *how to reason compositionally* about programs, or how to prove that a program is safe or correct.

Cosmo instantiates Iris, a generic framework for building concurrent separation logics. Iris provides the logical infrastructure, assertions, ghost state, invariants, weakest preconditions, and proof rules, while BaseCosmo supplies the Multicore OCaml–specific ingredients.

To make this precise, BaseCosmo introduces assertions such as points-to assertions and a valid-view assertion. Points-to assertions describe ownership and knowledge of individual memory locations, both nonatomic and atomic, while the valid-view assertion captures the global invariant relating thread views to the store. Atomic points-to assertions additionally account for the fact that atomic locations store a view (V), which can later be merged into a thread's view (W), explicitly propagating visibility.

Hoare triples[8] are the interface through which all of this reasoning is exposed. A triple (P, (e, W), Φ) states that if the precondition (P) holds, then executing expression (e) in a thread with view (W) is safe, and if it terminates, it produces a value and an updated view satisfying the postcondition (Φ). Iris supplies the generic machinery for weakest preconditions, framing, and invariants, while BaseCosmo provides the memory-specific axioms that explain how reads, writes, allocations, and atomic operations affect ownership and views.

To see how the operational semantics are reflected in the logic, consider the rule Mem-NA-Write. Operationally, this rule states that writing a value $v$ to a nonatomic location $a$ extends the history $h = \sigma(a)$ with a fresh timestamp $t$ such that $W(a) < t$, producing a new history $h' = h[t \mapsto v]$, and updates the thread's view to $W[a \mapsto t]$. Semantically, this means that a write does not overwrite a location in place; rather, it creates a new write event in the history of $a$, ordered after all writes currently visible to the thread. The store $\sigma$ is updated to record this extended history, while the thread's view is advanced to reflect awareness of this newly created event.

Let's take another example, consider the rule Mem-AT-Write. Operationally, this rule describes writing a new value $v'$ to an atomic location $A$. Suppose the atomic store contains $\sigma(A) = (v, V)$, where $V$ is the view currently stored at $A$. The rule updates the store to $\sigma[A \mapsto (v', V')]$, where

$$V' = W' = W \sqcup V.$$

Here, $\sqcup$ denotes the join of views.

This union is indicative of the fact that atomic writes in Multicore OCaml has both *release* and *acquire* effects. Before performing the write, the thread may already have learned new information about nonatomic locations, represented by its current view $W$. Meanwhile, the atomic location may already carry visibility information $V$ from previous synchronizations. By computing $W \sqcup V$, the write ensures that both pieces of information are preserved and propagated.

Intuitively, the atomic location acts as a visibility carrier. Whatever the writing thread currently knows (its view $W$) is merged with whatever knowledge was already attached to the atomic location ($V$), and the result is stored back into the location. Future threads that read from $A$ will acquire this combined view.

A small example clarifies this propagation. Suppose thread $T_1$ performs:

$$a :=_{\text{na}} 42; \quad A :=_{\text{at}} 1.$$

The nonatomic write creates a new timestamp in the history of $a$, and updates $T_1$'s view to record awareness of this write. When $T_1$ subsequently writes to atomic location $A$, its current view $W$—which now includes the write to $a$—is merged into the view stored at $A$.

Now suppose another thread $T_2$ later performs:

$$\texttt{!at}\ A.$$

By rule Mem-AT-Read, $T_2$'s view becomes

$$W_2' = W_2 \sqcup V',$$

where $V'$ is the view stored at $A$. Since $V'$ already contains the write to $a$, $T_2$ now becomes aware of that write—even though $a$ itself is nonatomic.

**A Higher Level Logic : Cosmo**

While BaseCosmo provides a faithful translation of the Multicore OCaml operational semantics, it remains intentionally low-level. In particular, it exposes details such as histories, timestamps, and per-thread views directly in assertions. This precision is useful for soundness, but it makes reasoning cumbersome: even in data-race-free code, a nonatomic location appears to store an entire history rather than a single value, and every Hoare triple must explicitly mention the current thread's view (W).

On top of BaseCosmo, Cosmo offers a higher-level logic in which nonatomic locations can again be reasoned about

as if they store a single value, provided the program is data-race free while still remaining sound with respect to relaxed memory.

Instead of asserting a BaseCosmo proposition directly, a Cosmo assertion denotes a function from views to BaseCosmo assertions. Subjective assertions, such as *this thread has seen view (V)*, depend on the current thread's snapshot and therefore cannot be shared. Objective assertions are independent of any particular thread's view and can be placed inside invariants. This distinction explains why nonatomic points-to assertions in Cosmo are necessarily subjective: they assert not only that a value exists, but that the current thread is aware of the corresponding write.

Iris [9] is used for the proof construction. It provides the general machinery for weakest preconditions. Hoare triples [8] in Cosmo look familiar: a precondition ($P$) describes the resources required to execute an expression ($e$), and the postcondition $\Phi$ describes the resources obtained after execution.

### 6.3 Conclusion

This paper introduced Cosmo, a concurrent separation logic that enables formal reasoning about Multicore OCaml under its relaxed memory model. Unlike traditional program logics that assume sequential consistency, Cosmo is designed to reason directly about the realities of modern multicore execution, including delayed visibility, partial views, and explicit synchronization.

The broader goal of this paper was also to understand how the verification of concurrent programs evolves as we move closer to real systems. While Cosmo significantly advances the state of the art by enabling formal reasoning about Multicore OCaml under a relaxed memory model, it is not the end of the story. As indicated by the authors, several important directions remain open, including support for arrays, richer synchronization primitives such as the Domain API, and full verification of sophisticated lock-free data structures. More fundamentally, reasoning about programs that intentionally exploit data races on nonatomic locations remains an open challenge.

## 7 Reagents: expressing and composing fine-grained concurrency [11][12]

### 7.1 Motivation

Following Herlihy et al. [6], a method is *wait-free* if every call is guaranteed to finish in a finite number of steps. It is *lock-free* if it ensures only that some thread always makes progress. A synchronization technique is *obstruction-free* if it guarantee progress for any thread that eventually executes in isolation. Obstruction free is strong enough to avoid the problems associated with locks, but it is weaker than previous nonblocking properties vis-a-vis lock-freedom and wait-freedom while wait-free, though they offer the strongest promise, are difficult to design and reason about.

Lock-free data structures are particularly useful because they eliminate the possibility of system-wide blocking due to stalled or failed threads. Unlike lock-based techniques, they avoid deadlock, priority inversion, and convoying effects, thereby improving robustness and scalability under contention. In highly concurrent systems, this guarantees global system progress even if individual threads are delayed, preempted, or crash.

Modern libraries such as `java.util.concurrent` provide highly optimized lock-free stacks, queues, and maps, yet these structures are not *composable*. For instance, performing `x = A.pop(); B.push(x);` is not atomic; between the two calls, another thread may observe inconsistent intermediate state. Extending the library with ad hoc methods like `popAndPush` leads to API explosion and violates a key abstraction principle: libraries should expose composable building blocks, not bespoke combinations. In traditional lock-free programming, the developer must explicitly orchestrate CAS loops, retries, and memory interactions, thereby controlling the low-level protocol. Reagents address this gap by introducing a new execution model. A reagent, written `Reagent`$[A, B]$, is a *description* of a concurrent atomic interaction: given input $A$, it performs coordinated memory updates and synchronization to produce $B$, while abstracting away CAS retries, transient failures, and commit protocols. By treating atomic operations as first-class composable values, reagents allow developers to declaratively specify interactions while the runtime enforces lock-free coordination. In doing so, they reconcile scalability with composability, avoiding global locks, heavyweight transactional memory, and abstraction-breaking APIs.

### 7.2 Proposed Solution

Reagents attempt to reconcile two historically distinct models of concurrency. The first model is the *shared-state paradigm*, in which threads coordinate by reading and writing shared memory locations using atomic primitives such as compare-and-swap (CAS). In this setting, correctness depends on carefully maintaining invariants over mutable state, and progress is achieved through retry loops that repeatedly attempt atomic updates. Lock-free stacks and queues are canonical examples of this approach.

The second model is the *message-passing paradigm*, in which threads do not interact by sharing memory directly but instead exchange values over channels. Synchronization is achieved through structured communication rather than low-level atomic instructions. This approach emphasizes clarity and composability, as communication events themselves become the unit of coordination.

Reagents unify these traditions by treating atomic memory updates as first-class synchronization events. In effect, they allow CAS-based interactions from the shared-state world to

be composed in a manner similar to communication events in the message-passing world.

### 7.3 Algebra of Reagents

Reagents introduce a small but expressive algebra for composing concurrent atomic interactions. At its core are three combinators: sequencing ($r_1$ ; $r_2$), parallel conjunction ($r_1$ * $r_2$), and choice ($r_1$ | $r_2$). Sequencing composes two reagents transactionally, ensuring that the effect of $r_2$ logically follows $r_1$ within a single atomic interaction. Parallel conjunction requires that both $r_1$ and $r_2$ be enabled and commit together, forming a coordinated atomic update across multiple memory locations. Choice allows the system to attempt multiple interactions and commit whichever becomes enabled first, reminiscent of selective communication. Unlike Software Transactional Memory (STM), where an `atomic` block tracks all reads and writes within its scope and validates them during commit, reagents expose only explicitly declared atomic updates; speculative reads remain invisible and do not participate in global validation. Composition therefore occurs at well-defined commit boundaries rather than at the level of arbitrary memory accesses. For example, composing `pop`($A$) ; `push`($B$) yields an atomic transfer between two lock-free stacks without introducing a global transaction that tracks every intermediate read. Similarly, $r_1$ | $r_2$ allows a program to proceed with whichever operation becomes feasible first, and $r_1$ * $r_2$ coordinates multiple atomic updates as a single interaction. In this way, reagents provide algebraic composition of lock-free operations while avoiding the heavyweight bookkeeping characteristic of STM.

### 7.4 Implementation Overview

Reagents execute via a conceptually two-phase protocol [12]. When a reagent is invoked (via the ! method), it attempts to *react*. Reaction consists of (1) a *collection phase*, during which the runtime builds up a `Reaction` object containing the atomic updates (e.g., CAS operations), messages, and post-commit actions; and (2) a *commit phase*, during which the collected updates are atomically applied.

A failure during the first phase (i.e., failure to build the desired reaction) is classified as a *permanent failure*. This corresponds to logical impossibility under current conditions (e.g., popping from an empty stack with no alternative branch). Retrying would not help; the reagent must block until external state changes. By contrast, a failure during the commit phase is a *transient failure*: the reaction was valid, but interference from another thread prevented atomic commitment. In this case, the system retries.

### 7.5 Reagent Syntax and Semantics

The Reagents library provides a distinct syntax for expressing fine-grained concurrency, centered around the composition of atomic transactions. A Reagent,in Scala syntax $R[A, B]$ ,is not a simple function from $A$ to $B$, but a description of an atomic transaction that transforms an input of type $A$ into a result of type $B$.

The core operation for executing a reagent is the exclamation mark (!), which initiates the transaction:

```
result = reagent ! input
```
(11)

This invocation triggers the aforementioned two-phase process: *accumulation*, where the transaction's effects (such as Compare-and-Swap operations) are collected into a `Reaction` object, and *commit*, where these effects are atomically applied.

Reagents are composed using combinators that can be thought of as bind operations. The sequencing combinator (») chains reagents together:

$$R_1 \gg R_2 \tag{12}$$

This expression implies a dependency: "Attempt $R_1$; if successful, use its result as input for $R_2$. If either fails, the entire transaction aborts."

Internally, this behavior is implemented via the `tryReact` method:

```
def tryReact(a: A, rx: Reaction, offer:
    Offer[B]): Any
```

Here, `rx` represents the accumulated state of the transaction so far—the pending CAS operations and message sends.

### 7.6 The Necessity of Continuation-Passing Style (CPS)

The sequential composition $R_1 \gg R_2$ presents a fundamental challenge: $R_1$ cannot simply return a value to $R_2$ because the transaction is not yet complete. The system must maintain the ability to abort, retry, or block the entire chain based on future conditions. This requirement necessitates *Continuation-Passing Style (CPS)*.

**7.6.1 Inverting Control Flow.** In standard direct-style programming, a function returns a result to its caller. In CPS, a function receives the "rest of the computation" (the continuation) as an argument and decides how to proceed.

The `tryReact` method embodies this principle. The `rx` argument effectively serves as the continuation. Instead of computing a final result, a reagent transforms the current continuation:

This allows the library to build up the transaction step-by-step without committing any side effects until the entire chain is ready.

**7.6.2 Explicit Failure and Retry.** CPS is structurally required to handle the non-deterministic nature of concurrent programming. A reagent does not just succeed or fail; it has three possible outcomes:

1. **Success:** The reagent extends the continuation with its operations and passes control to the next reagent.

```
// The tryReact function transforms an input
     and a continuation // into either a
new continuation or a failure. // // Syntax
    Mapping: // 1. × becomes (A,
B): In Scala, a product type is a tuple/
    argument list. // 2. ⟶ becomes =>:
The function arrow in Scala is =>. // 3. ∨
    becomes |: In Scala 3, A | B is
a Union Type, mapping "Success OR Failure".

type TryReact = (A, CurrentContinuation) =>
    NewContinuation | Failure
```

**Figure 12.** Conceptual type signature of TryReact

2. **Transient Failure (Retry):** Interference occurred (e.g., a CAS failed). The continuation is discarded, and the transaction restarts.
3. **Permanent Failure (Block):** The operation cannot proceed (e.g., waiting on an empty queue). The thread suspends execution.

By using CPS, the Reagents library treats the execution path as a first-class value that can be suspended, discarded, or re-executed. This explicit management of the continuation allows for the composition of complex, lock-free synchronization primitives—such as the kCAS protocol—that would be impossible to express using standard return values.

### 7.7 Conclusion

Reagents present an abstraction that blends shared-state concurrency with message passing in a compositional and programmable manner. By structuring fine-grained synchronization as composable with explicit commit boundaries they offer a transactional view of lock-free data structures.

The paper positions reagents as serving both concurrency experts and concurrency users: experts gain expressive abstractions for recurring synchronization patterns, while users can extend and compose these libraries without intimate knowledge of the underlying algorithms.

At the same time, the author acknowledge significant open directions, notably the need for a formal operational semantics and the development of substantial concurrency libraries. These future directions align closely with ongoing research in automated linearizability verification and formal reasoning about shared-state interactions under weak memory models. For this reason, the paper is especially relevant: it not only proposes a powerful programming abstraction, but also outlines a research agenda that connects directly to the broader goal of automatic verification of concurrent systems.

## 8 Conclusion and Research Outlook

Writing this report has been, in a sense, an exercise in cartography. The thread running through all of it is deceptively simple: how do we reason about the correctness of concurrent programs? Linearizability answers this with a clean semantic contract ; operations appear to take effect instantaneously, at a single point in time. It restores sequential intuition to a world of interleaved execution and grants compositionality almost for free. But as the MCAS case study showed, even stating where an operation *takes effect* becomes deeply nontrivial once helping and non-local linearization points enter the picture.

Instrumentation and Rely-Guarantee reasoning were the first attempt to make this tractable. Rather than quantifying over all concurrent histories, one reduces linearizability to a safety property, verified locally under interference summaries. When Zoo entered the picture, that assumption evaporated. Verifying actual OCaml code means confronting the full complexity of a real language: physical equality, compiler sharing, and the realization that correctness can depend not only on algorithmic structure but on guarantees about memory representation and object identity.

Once sequential consistency is abandoned ; as it must be for any serious reasoning about modern multicore OCaml ; atomicity can no longer be treated as a clean global concept. Every correctness argument must now account simultaneously for algorithm design, release/acquire annotations, and the guarantees (and gaps) of the underlying memory model. Cosmo makes this concrete, and the added complexity it introduces feels less like a complication and more like an honest accounting of what is out there in real world systems.

Reagents approaches the problem from a different angle entirely. Rather than verifying fine-grained synchronization after the fact, it restructures concurrency itself around explicit commit boundaries.

Having worked through all of these papers, what I am left with is not uncertainty about the field, but a clearer sense of where the interesting problems are. Concurrency verification is not a single discipline: it is a negotiation between correctness conditions and language semantics. Each of the six works covered in this report sharpens a different facet of that negotiation. My own research direction is still taking shape, but the vocabulary and the conceptual tools are now in place. Whatever problem I eventually commit to, I expect the ingredients for tackling it to already exist ; distributed across these bodies of work and waiting to be recombined.

## References

[1] Clément Allain and Gabriel Scherer. Zoo: A framework for the verification of concurrent ocaml 5 programs using separation logic. *Proc. ACM Program. Lang.*, 10(POPL), January 2026.

[2] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of ion-implanted mosfets with very small physical dimensions. *IEEE Journal of Solid-State*

*Circuits*, 9(3):256–268, 1974.

[3] Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. Efficient multi-word compare-and-swap. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 221–232, Shenzhen, China, 2013. ACM.

[4] Timothy L. Harris and Keir Fraser. A practical multi-word compare-and-swap operation. *Distributed Computing*, 15(1):257–271, 2002.

[5] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, page 265–279, Berlin, Heidelberg, 2002. Springer-Verlag.

[6] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, Providence, Rhode Island, 2003. IEEE Computer Society.

[7] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(2):463–492, 1990.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[9] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2017.

[10] Glen Mével, Jacques-Henri Jourdan, and François Pottier. Cosmo: a concurrent separation logic for multicore ocaml. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.

[11] Aaron Turon. Reagents: expressing and composing fine-grained concurrency. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 157–168, New York, NY, USA, 2012. Association for Computing Machinery.

[12] Aaron Turon. *Understanding and Expressing Scalable Concurrency*. PhD thesis, Northeastern University, 2013.

[13] Viktor Vafeiadis. Automatically proving linearizability. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 450–464, Edinburgh, Scotland, 2010. Springer.

[14] Viktor Vafeiadis. Rgsep action inference. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 357–371, 2010.