# Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic

JAEHWANG JUNG, KAIST, Republic of Korea
JANGGUN LEE, KAIST, Republic of Korea
JAEMIN CHOI, KAIST, Republic of Korea
JAEWOO KIM, KAIST, Republic of Korea
SUNHO PARK, KAIST, Republic of Korea
JEEHOON KANG, KAIST, Republic of Korea

Formal verification is an effective method to address the challenge of designing correct and efficient concurrent data structures. But verification efforts often ignore *memory reclamation*, which involves nontrivial synchronization between concurrent accesses and reclamation. When incorrectly implemented, it may lead to critical safety errors such as use-after-free and the ABA problem. Semi-automatic safe memory reclamation schemes such as hazard pointers and RCU encapsulate the complexity of manual memory management in modular interfaces. However, this modularity has not been carried over to formal verification.

We propose modular specifications of hazard pointers and RCU, and formally verify realistic implementations of them in concurrent separation logic. Specifically, we design abstract predicates for hazard pointers that capture the meaning of *validating* the protection of nodes, and those for RCU that support *optimistic traversal* to possibly retired nodes. We demonstrate that the specifications indeed facilitate modular verification in three criteria: compositional verification, general applicability, and easy integration. In doing so, we present the first formal verification of Harris's list, the Harris-Michael list, the Chase-Lev deque, and RDCSS with reclamation. We report the Coq mechanization of all our results in the Iris separation logic framework.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Program verification**; **Concurrent algorithms**.

Additional Key Words and Phrases: safe memory reclamation, separation logic, Iris

**ACM Reference Format:**
Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. 2023. Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 251 (October 2023), 35 pages. https://doi.org/10.1145/3622827

## 1 INTRODUCTION

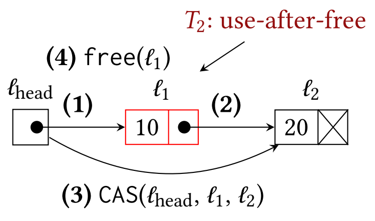
It is challenging to design correct and efficient *concurrent data structures*. An effective method to address the challenge is *formal verification*, which not only increases our confidence in the correctness of the algorithm but also help us improve it. As such, various concurrent data structures have been formally verified.

Authors' addresses: Jaehwang Jung, jaehwang.jung@kaist.ac.kr, KAIST, Daejeon, Republic of Korea; Janggun Lee, janggun.lee@kaist.ac.kr, KAIST, Daejeon, Republic of Korea; Jaemin Choi, jaemin.choi98@kaist.ac.kr, KAIST, Daejeon, Republic of Korea; Jaewoo Kim, jaewoo.kim@kaist.ac.kr, KAIST, Daejeon, Republic of Korea; Sunho Park, sunho.park@kaist.ac.kr, KAIST, Daejeon, Republic of Korea; Jeehoon Kang, jeehoon.kang@kaist.ac.kr, KAIST, Daejeon, Republic of Korea.

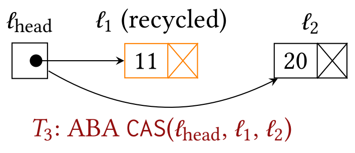
2475-1421/2023/10-ART251
https://doi.org/10.1145/3622827

```
C1    fun pop(st):
C2 R    rcu_lock(tid)
C3      loop:
C4 N      h := (*st).head
C4 H      h := protect(tid, &(*st).head)
C4 R      h := (*st).head
C5        if h == NULL:
C6 H        unprotect(tid)
C6 R        rcu_unlock(tid)
C7          return None
C8        x := (*h).data; n := (*h).next
C9        if CAS(&(*st).head, h, n):
C10 N       // free(h) incurs errors
C10 H       retire(h); unprotect(tid)
C10 R       retire(h); rcu_unlock(tid)
C11         return Some(x)
```

(a) pop() code without reclamation (red lines with N), with hazard pointers (green lines with H), and with RCU (blue lines with R).



(b) $T_2$ accesses $\ell_1$ already reclaimed by $T_1$.



(c) $T_3$ uses stale values for CAS.

Fig. 1. Problems of immediately reclaiming detached block illustrated in Treiber's stack [Treiber 1986].

However, the existing verifications of concurrent data structures often ignore *memory reclamation*, which involves nontrivial synchronization between concurrent accesses and reclamation of the same memory block. For instance, consider Treiber's concurrent stack [Treiber 1986] presented in Fig. 1a. (For now, ignore those lines marked with H or R.) A Treiber's stack is essentially a linked list of elements with its head being the stack top. Fig. 1b illustrates the procedure of the pop() method. When a thread invokes pop() of a stack at address st, it **(1)** reads the pointer to the first block $\ell_1$ from st's head field (line C4); **(2)** if $\ell_1$ is NULL, returns None (C5-7); **(3)** otherwise, reads $\ell_1$'s data and the pointer to its next block $\ell_2$ (line C8); and **(4)** detaches $\ell_1$ by performing compare-and-swap (CAS) on head that atomically replaces $\ell_1$ with $\ell_2$ (line C9). If successful, pop() returns $\ell_1$'s data (line C11), and otherwise, it retries from the beginning (line C3). **(5)** To avoid memory leaks, it should eventually reclaim the detached block $\ell_1$. What if pop() immediately reclaims $\ell_1$ at line C10? Then the following critical errors would occur:

- ***Use-After-Free*** (Fig. 1b): Suppose threads $T_1$ and $T_2$ concurrently invoke pop() to take $\ell_1$. Then a use-after-free error would occur in the following scenario: $T_2$ loads the pointer $\ell_1$ (line C4); $T_1$ detaches and reclaims $\ell_1$ (line C10); and $T_2$ accesses $\ell_1$ (line C8) that is already reclaimed by $T_1$.
- ***ABA Problem*** (Fig. 1c): Suppose another thread $T_3$ concurrently invokes pop() to take $\ell_1$. Then its result would not be stack-like (*i.e.*, not *linearizable* [Herlihy and Wing 1990]) in the following scenario: $T_3$ accesses $\ell_1$ and the node's value 10 (line C4-8); $T_1$ invokes pop() twice, each of which detaches and reclaims $\ell_1$ (resp. $\ell_2$) and returns 10 (resp. 20); $T_1$ invokes push(11), which allocates a block that happens to be the recycled $\ell_1$, and inserts $\ell_1$ with the new value 11 at the top; $T_3$ successfully performs a CAS($\ell_{\text{head}}$, $\ell_1$, $\ell_2$) (line C9), and returns 10. This behavior is invalid for a stack because $T_3$'s pop() returns 10 which is already popped by $T_1$. At the

high level, this error occurs because $T_3$ cannot distinguish $\ell_1$ between the older and the newer allocations.

To prevent such errors, `pop()` should defer the deallocation of $\ell_1$ until all the other threads have finished referencing it, so that each thread's accesses are safe and the logical identities of pointers do not change within an operation. The simplest solution is to use tracing garbage collectors (GC) that automatically reclaim memory blocks when it is safe to do so. While programmers do not need to care about reclamation when using GC, it is usually unavailable for low-level systems, and even if available, it may not be the best option due to its performance overhead. However, it is difficult to design a correct and efficient manual memory management method for each case.

## 1.1 Modular Implementation of Memory Reclamation

To alleviate the difficulty of manual memory management, various semi-automatic *safe memory reclamation schemes* (*SMR schemes* from now on) have been proposed, *e.g.*, hazard pointers [Michael et al. 2023; Michael 2004], RCU [Fraser 2004; Harris 2001; Hart et al. 2007; McKenney and Slingwine 1998; McKenney et al. 2023], and their hybrids [Alistarh et al. 2017, 2018; Brown 2015; Kang and Jung 2020; Nikolaev and Ravindran 2020, 2021; Sheffi et al. 2021; Singh et al. 2021; Wen et al. 2018]. SMR schemes modularize memory management by separating concerns between data structure operation and reclamation. They provide an abstraction layer consisting of **(1)** a function to *protect* pointers to prevent their deallocation; and **(2)** a function to *retire* pointers so that they can later be reclaimed when no threads are protecting them. Concurrent data structures only need to use these functions (without understanding their implementation) to protect pointers before accessing and to retire pointers after detaching. Then the synchronization between the protection and reclamation is automatically undertaken by the SMR scheme.

***Hazard Pointers.*** For instance, hazard pointers ensures safe memory reclamation in Treiber's stack as follows (green lines marked with `H` in Fig. 1a). **(1)** At line C4, we replace the load instruction `(*st).head` of the head pointer with a function call `protect(tid, &(*st).head)` that loads a *protected pointer* `h` to the head block for the current thread `tid`.[1] This ensures that the thread can safely dereference `h` at line C8. **(2)** At lines C6 `H` and C10 `H`, before returning from the function, we invoke `unprotect(tid)` to revoke the protection of the pointer `h`. **(3)** At line C10 `H`, after detaching `h` from the stack, we invoke `retire(h)` to schedule the reclamation of `h`. The `protect()` function achieves its goal by publishing the pointer value to the thread's *protected pointer slot*, and the reclaimer frees a retired pointer only if it is not written in any of those slots.

***RCU.*** For another example, RCU provides a coarse-grained protection for *all* pointers accessible inside a *critical section* delimited by `rcu_lock(tid)` and `rcu_unlock(tid)`. RCU ensures safe memory reclamation in `pop()` as follows (blue lines marked with `R` in Fig. 1a). **(1)** We first enter a critical section with `rcu_lock(tid)` (line C2 `R`). **(2)** All pointers obtained inside the critical section, *e.g.*, `h` at line C4 are protected in the critical section. Therefore, the accesses to `h` at line C8 are safe without any further action. **(3)** After detaching `h` from the stack (line C10 `R`), we call `retire(h)`. **(4)** Finally, before returning from `pop()` (lines C6 `R` and C10 `R`), we call `rcu_unlock(tid)` to exit the critical section and thus revoke the protection. RCU implements protection by deferring the reclamation of each retired pointer until all threads end their critical section in which the pointer may be accessed.

[1]For concise presentation, we use a simple version of hazard pointers in which each thread can protect a single pointer. In our formalized verification, we use the generalized version that allows each thread to protect an unbounded number of pointers. In that version, protection is identified by *slot ID* instead of thread ID.

## 1.2 Problem: Non-Modular Verification of Memory Reclamation

However, the modularity of the SMR schemes has not been carried over to formal verification of concurrent data structures with reclamation. Specifically, each of the state-of-the-art verification efforts [Gotsman et al. 2013; Tassarotti et al. 2015; Wolff 2021] lacks at least one of the following desired properties. **(1) Compositional verification**: Concurrent data structures and SMR schemes should be individually verified and then composed without understanding each other's implementation. **(2) General applicability**: The verification method should be powerful enough to verify a variety of concurrent data structures with SMR schemes. **(3) Easy integration**: The verification of concurrent data structures with SMR schemes should be easily adapted from the verification of the version without reclamation. With these criteria, we briefly discuss the existing verification efforts in the following (see §8 for more detail).

Gotsman et al. [2013] verified data structures integrated with hazard pointers and RCU using a temporal separation logic. They use temporal invariants enforcing that the pointers which "have been protected *since* they were reachable" are still allocated. However, their approach is *not compositional* as it tightly couples the implementation details of SMR schemes and data structures. Specifically, the two sub-propositions "protected" and "reachable" are about implementation details of SMR scheme and data structure, respectively. While this method seems to be *generally applicable* in principle, it is not demonstrated for a wide range of examples.

Tassarotti et al. [2015] verified a single-writer multi-reader linked list integrated with quiescent-state-based RCU [Desnoyers et al. 2012; Hart et al. 2007]. Their method models manual memory management purely in terms of ownership transfer, without relying on temporal logic. In addition, their verification assumes a more realistic *relaxed memory model* [Batty et al. 2011; Kang et al. 2017; Lahav et al. 2017] in which memory accesses can be reordered due to hardware and compiler optimizations, whereas most of the prior work assumes the sequentially consistent memory model [Lamport 1979]. However, their approach is *not compositional*, because they did not use a general specification of RCU to verify the linked list. Instead, they monolithically verified a linked list integrated with an RCU implementation using an invariant that tightly couples the operation history of the linked list and RCU internals and assumes the uniqueness of the writer thread. In addition, while the core idea of their method is *generally applicable*, it is not applied to other SMR schemes and data structures.

Meyer and Wolff [2019a,b]; Wolff [2021] developed an automatic linearizability checker for concurrent data structures with SMR schemes. Their verification is *compositional* as it is parametric over the specifications for each SMR scheme. Their method is *easy to integrate* as their verifier automatically checks whether a given linearizable concurrent data structure without reclamation can be adapted to that with reclamation. However, their method is *not generally applicable* because it relies on a linearizability checker that does not scale to sophisticated non-blocking data structures such as Harris's list [Harris 2001].

## 1.3 Contributions: Modular Verification of Memory Reclamation

We propose modular specifications of hazard pointers and RCU, formally verify realistic implementations of them, and demonstrate that the specifications indeed facilitate modular verification of memory reclamation. Specifically, we make the following contributions.

- In §2, we describe the challenges in designing and verifying modular specifications of hazard pointers and RCU, which include the subtleties in *validating* protection of nodes in hazard pointers, and RCU's support for *optimistic traversal* to possibly retired nodes. In doing so, we review the necessary technical background on hazard pointers, RCU, and separation logic-based verification of concurrent data structures.

```
C11 fun protect(tid, src):
C12   p := *src
C13   loop:
C14     protected[tid] := p
C15     p' := *src
C16     if p' == p:
C17       return p'
C18     p := p'

C21 fun unprotect(tid):
C22   protected[tid] := NULL
```

```
C31 fun retire(p):
C32   retired.push(p)
C33   if /* some condition */:
C34     do_reclamation()

C41 fun do_reclamation():
C42   for r in retired.pop_all():
C43     if r in protected:
C44       retired.push(r)
C45     else:
C46       free(r)
```

Fig. 2. A simplified implementation of hazard pointers.

- In §3, we propose a specification of hazard pointers. The key idea lies in designing abstract predicates to precisely capture the meaning of validating protection. For presentation purposes, we make a simplifying assumption that a memory block's contents are immutable in this section and lift the assumption in §5.
- In §4, we discuss the key ideas for verifying the above specification.
- In §5, we generalize the above specification to mutable memory blocks.
- In §6, we propose a specification of RCU. The key idea lies in designing abstract predicates to precisely capture the guarantees provided by a critical section and to encapsulate the reasoning about the link structure for optimistic traversal.
- In §7, we evaluate the modularity of our specifications of hazard pointers and RCU with the three criteria discussed in §1.2. Specifically, we have *compositionally verified* realistic implementations of hazard pointers and RCU (based on Meta's Folly library [Meta 2023] and the non-blocking epoch-based algorithm by Parkinson et al. [2017], respectively) and the functional correctness of 9 non-blocking data structures with reclamation (*general applicability*). We observe that, compared to their counterparts without reclamation, the verification overhead is roughly proportional to the implementation overhead (*easy integration*).

In §8 and 9, we conclude with related and future work. In the supplementary material [Jung et al. 2023], we report the Coq mechanization of all our results in the Iris separation logic framework [Iris Team 2023b; Jung et al. 2018, 2015; Krebbers et al. 2017].

## 2 BACKGROUND AND CHALLENGES

### 2.1 Hazard Pointers

Fig. 2 shows a simplified implementation of hazard pointers. To ensure safe use of protected pointers, it defers the reclamation of retired pointers until they are no longer protected by any threads.

On the one hand, `retire(p)` adds `p` to the *retired pointer list* (`retired`, line C32). When some implementation-specific conditions are met—*e.g.*, the number of retired pointers exceeds a certain threshold (line C33), `retire()` calls `do_reclamation()` to reclaim those retired pointers that are not currently protected (line C34). The `do_reclamation()` function first atomically removes all pointers from the retired pointer list (line C42). Then it checks if each pointer is in the *protected pointer list* (`protected`, line C43). If so, the pointer is added back to the retired pointer list (line C44). Otherwise, the pointer is reclaimed (line C46).

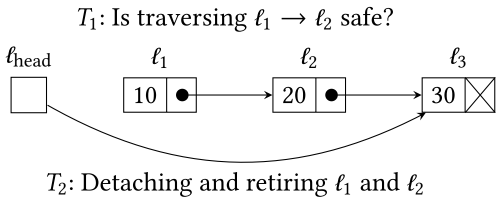

$T_1$: Is traversing $\ell_1 \to \ell_2$ safe?
$\ell_{\text{head}}$
$\ell_1$
$\ell_2$
$\ell_3$
10
20
30
$T_2$: Detaching and retiring $\ell_1$ and $\ell_2$

Fig. 3. Traversing possibly retired nodes.

On the other hand, `protect(tid, src)` loads a pointer, say `p`, from `src` (line C12), and stores `p` in the thread `tid`'s slot of the protected pointer list (line C14). However, it is not yet safe to use `p` because it could have already been retired and then freed by other threads. Therefore, one should *validate* that the pointer is not retired. This is usually done by checking that the memory block is still *reachable* from the data structure, assuming that only detached (*i.e.*, unreachable) blocks are retired. For example, `protect()` validates the protection by checking whether `src` still points to `p` (line C15-16).[2] If validation fails, `protect()` retries from the beginning (line C13-18).

***Verification Challenges.*** Validation makes hazard pointers more difficult to apply (and verify) than it seems in two aspects.

**(1)** The requirement for validation is fundamentally incompatible with *optimistic traversal* to possibly retired nodes, a common optimization pattern in concurrent data structure design. For instance, in Fig. 3 where $T_2$ detaches and retires $\ell_1$ and $\ell_2$, is it safe for $T_1$ to traverse from $\ell_1$ to $\ell_2$? This is unsafe when using hazard pointers because $\ell_2$ could have been retired and reclaimed before $T_2$ tried protecting it. Then the only reasonable option for $T_2$ is to restart the traversal from $\ell_{\text{head}}$, possibly incurring performance degradation. Therefore, when applying hazard pointers, the data structure must be modified to handle such scenarios. For example, the Harris-Michael list [Michael 2002] is an adaptation of Harris's lock-free list [Harris 2001] that forgoes optimistic traversal for compatibility with hazard pointers.

**(2)** But still, how can $T_1$ detect that $\ell_2$ may have been retired? In general, `protect()`'s validation does not work: `protect((*`$\ell_1$`).next)` would return $\ell_2$ despite that it might have been already retired. Therefore, sophisticated data structures resort to custom validation strategies that exploit the data structures' complex invariant. For example, the Harris-Michael list requires collaboration from the deleting thread: before $T_2$ detaches $\ell_1$, it first *marks* the link $\ell_1 \to \ell_2$ by setting the least significant bit (LSB) of the pointer value. Then, if $T_1$ sees that $\ell_1 \to \ell_2$ is marked, the validation of $\ell_2$ fails conservatively.[3] This is because if $\ell_2$ is detached and yet $\ell_1$ still links to $\ell_2$, then $\ell_1$ must have been detached too, by the definition of "detaching".

These subtleties are sometimes misunderstood even by experienced programmers and lead to critical bugs [Anderson et al. 2021]. In §3 and §5, we will capture such subtleties within a powerful yet modular specification of hazard pointers.

[2]The comparison of pointers at this point may involve an invalid (dangling) pointer, which is an undefined behavior in C/C++'s provenance-based pointer semantics. Since the comparison of invalid pointers is unavoidable in SMR schemes and many other low-level concurrent algorithms, there is a proposal to introduce a special pointer type that is exempted from this strict semantics [McKenney et al. 2021]. We assume the proposed lenient semantics.
[3]The marking process, called *logical deletion*, is required even when hazard pointer is not used. This is a common technique in concurrent data structure design for synchronizing concurrent updates in linked data structures. Hazard pointers piggyback on this mechanism for validation.

## 2.2 RCU

RCU is more straightforward to use than hazard pointers thanks to critical section-based protection: a pointer is protected throughout a critical section if it were not retired before the beginning of the critical section [McKenney et al. 2023]. Specifically, in Fig. 3, it is safe to perform optimistic traversal from $\ell_1$ to $\ell_2$ because RCU's protection condition implies that *all* memory blocks reachable by traversing the data structure—including just retired $\ell_2$—are protected. Therefore, users can seamlessly integrate RCU into existing data structures such as Harris's list without worrying about validation and restarting.

***Verification Challenges.*** However, the formal verification of the safety of optimistic traversal using RCU is challenging. To guarantee the protection of reachable blocks, one should deduce that they were not retired before the beginning of the critical section by reasoning about the history of updates of links among memory blocks and their retirement. This has been tackled by Tassarotti et al. [2015] for a fixed data structure with the simplifying assumption that updates are done by a single writer thread.[4] In §6, we will generalize their approach and encapsulate this reasoning in a modular specification for RCU.

## 2.3 Verification of Treiber's Stack without Reclamation

We review a separation logic-based verification of Treiber's stack without reclamation. We will adapt this proof to the version with hazard pointers (§3) and RCU (§6).

***Separation Logic Primer.*** First, we briefly overview the fragment of the Iris separation logic we will be using. Some concepts not explained here will be gradually introduced along the way. We refer the reader to Jung et al. [2018, §2] for a more detailed overview.

$$\begin{array}{rll} P, Q \in \mathit{iProp} ::= & \phi \mid P \land Q \mid P \lor Q \mid P \Rightarrow Q \mid \exists x.\, P \mid \forall x.\, P \mid \ldots & \text{higher-order logic} \\ & \ell \overset{q}{\mapsto} v \mid \gamma \overset{q}{\Mapsto} v \mid \boxed{P} \mid \ldots & \text{separation logic resources} \\ & P * Q \mid P \mathrel{-\!\!*} Q \mid P \Rrightarrow\!\!\ast\; Q \mid \Box P \mid \ldots & \text{separation logic connectives} \\ & \{P\}\, e\, \{v.\, Q(v)\} \mid \langle x.\, P(x)\rangle\, e\, \langle v.\, Q(x,v)\rangle \mid \ldots & \text{program logic} \end{array}$$

*iProp* is the type of Iris's separation logic proposition. Based on higher-order logics, *iProp* includes usual propositions and connectives. But more importantly, an *iProp* asserts ownership of resources. For example, the *points-to* assertion $\ell \mapsto v$ (shorthand for $\ell \overset{1}{\mapsto} v$, explained later), represents the exclusive ownership of a memory block at location $\ell$ that contains a value $v$. Such resources can be combined with the separating conjunction ($*$). For instance, the stack illustrated in Fig. 1b before `pop()` owns the following resource (where $\{\texttt{.field} = v, \ldots\}$ is a struct value):

$$\ell_{\text{head}} \mapsto \{\texttt{.head} = \ell_1\} * \ell_1 \mapsto \{\texttt{.data} = 10, \texttt{.next} = \ell_2\} * \ell_2 \mapsto \{\texttt{.data} = 20, \texttt{.next} = \texttt{NULL}\}\ .$$

A *Hoare triple* of the form $\{P\}\, e\, \{v.\, Q(v)\}$ asserts that given resources satisfying the *precondition* $P$, program $e$ evaluates to $v$ without errors such as null pointer dereferences, and returns the resource satisfying the *postcondition* $Q(v)$. For example, the owner of $\ell \mapsto v$ can read from, write to, and reclaim $\ell$:

$$\begin{array}{lll} \text{(PointsTo-Read)} & \text{(PointsTo-CAS-Success)} & \text{(PointsTo-Free)} \\ \{\ell \mapsto v\}\, {*\ell}\, \{v.\, \ell \mapsto v\} & \{\ell \mapsto v\}\, \texttt{CAS}(\ell, v, w)\, \{\texttt{true}.\, \ell \mapsto w\} & \{\ell \mapsto \_\}\, \texttt{free}(\ell)\, \{\mathsf{True}\}\ . \end{array}$$

However, in the stack example, threads do not exclusively own the resources, but share them. Iris provides the *invariant* assertion of the form $\boxed{I}$ to describe a shared resource $I$ that can be accessed

[4]Though note that Tassarotti et al. [2015] assume a more realistic relaxed memory model while we focus on the SC model.

by multiple threads. In particular, invariants provide the *access rule*:

$$\text{(I{\small NV}-A{\small CC})}\quad \frac{\{I * P\}\, e\, \{I * Q\} \qquad e \text{ is physically atomic}}{\boxed{I} \vdash \{P\}\, e\, \{Q\}}$$

The rule says that, if $\boxed{I}$ holds, then during the duration of an atomic instruction $e$ that evaluates in a single step, the program can temporarily *open* the invariant to use the content $I$, and it should *close* the invariant by reestablishing $I$ after the execution.[5] Invariant assertions are *duplicable*, *i.e.*, $\boxed{I} \dashv\vdash \boxed{I} * \boxed{I}$ (entailment in both directions), so they can be freely shared.[6] In the stack example, threads share the invariant containing the stack nodes.

***Specification.*** In this paper, we focus on proving the following simple safety specification of pop(), where IsStack($st$) is an invariant that describes the shared resources in a Treiber's stack located at $st$ (defined below).

$$\mathsf{IsStack}(st) \vdash \{\mathsf{True}\}\, \mathsf{pop}(st)\, \{v.\, \mathsf{True}\}\ .$$

In our Coq development, we prove a much stronger specification based on *logically atomic Hoare triples* [da Rocha Pinto et al. 2014; Jacobs and Piessens 2011; Jung 2019; Jung et al. 2015; Svendsen and Birkedal 2014]:

$$\left\langle xs.\, \mathsf{Stack}(st, xs) \right\rangle \mathsf{pop}(st) \left\langle v.\, \exists xs'.\, \mathsf{Stack}(st, xs') * \bigvee \begin{cases} v = \mathsf{None} \land xs = xs' = [] \\ \exists x.\, v = \mathsf{Some}(x) \land xs = x :: xs' \end{cases} \right\rangle .$$

A logically atomic triple of the form $\langle x.\, P(x)\rangle\, e\, \langle v.\, Q(x, v)\rangle$ is a special Hoare triple (indicated by ⟨angle brackets⟩) that says $e$ behaves *as if* it were an atomic instruction. Specifically, it reads: at $e$'s *commit point* (an atomic instruction inside $e$), $e$ takes $P(x)$ as precondition, evaluates to $v$, and returns postcondition $Q(x, v)$. Logically atomic triples encode linearizability [Herlihy and Wing 1990] in program logic. For example, the above specification implies that pop() is a linearizable implementation of stack's pop method: at the commit point, *i.e.*, the *linearization point*, pop() atomically transforms the stack's state from $xs$ to $xs'$, and the result is either None if $xs$ was empty (the first disjunct) or Some($x$) where $x$ is the head of $xs$ (the second disjunct).

Logically atomic triples can be used with the following rules.

$$\text{(LAT-H{\small OARE})}\quad \frac{\langle P\rangle\, e\, \langle Q\rangle}{\{P\}\, e\, \{Q\}} \qquad\qquad \text{(LAT-I{\small NV}-A{\small CC})}\quad \frac{\langle I * P\rangle\, e\, \langle I * Q\rangle}{\boxed{I} \vdash \langle P\rangle\, e\, \langle Q\rangle}$$

Clearly, a logically atomic triple implies the ordinary counterpart (LAT-Hoare). More importantly, logical atomicity of $e$ means that $e$ can access invariants as if it were an atomic instruction (LAT-Inv-Acc). Therefore, specifications with logically atomic triples allow clients to atomically access the current state of the object under question, enabling them to build sophisticated protocols around them. We take advantage of this for specifying SMR schemes in §3 and 6. However, *proving* logically atomic triples involves many technicalities orthogonal to this work. So, for a concise presentation, we discuss the details in the appendix [Jung et al. 2023].

[5] For a concise presentation, we omit Iris's mechanisms for preventing opening the same invariant twice.

[6] More precisely, invariants are *persistent*. Persistent propositions represent some knowledge that holds forever, rather than asserting ownership of resources.

***Fractional and Leaking Points-To Predicates.*** We now formally define IsStack using a predicate describing the valid states of Treiber's stack.

$$\mathsf{IsStack}(st : Loc) \coloneqq \boxed{\exists h : Loc,\ xs : List(Val).\ st \mapsto \{.\mathsf{head} = h\} * \mathsf{LinkedList}(h, xs)}$$

$$\mathsf{LinkedList}(h, xs) \coloneqq \bigvee \begin{cases} h = \mathtt{NULL} \land xs = [] \\ \exists x, xs', n.\, xs = x :: xs' \\ \quad * \, h \mapsto^{?} \{\mathtt{.data} = x,\ \mathtt{.next} = n\} * \mathsf{LinkedList}(n, xs') \end{cases}$$

As expected, the content of IsStack is a generalization of the stack resource discussed above. In particular, LinkedList$(h, xs)$ consists of the points-to assertions of every memory block in the linked list of elements $xs$ starting from the head node $h$.

However, notice that LinkedList uses a variant of points-to assertion $\ell \mapsto^{?} v$ which we call *leaking points-to*. $\ell \mapsto^{?} v$ represents a read-only permission to an immutable location $\ell$ that never gets reclaimed. The notation $\ell \mapsto^{?} v$ is a shorthand for $\exists q.\, \ell \overset{q}{\mapsto} v$, the *fractional points-to* [Bornat et al. 2005; Boyland 2003] with some *unknown fraction* $q$. Fractional points-to allows splitting ownership into fractional parts that allow read accesses only and can be combined back to the full ownership $\ell \overset{1}{\mapsto} v$ to recover the right to write and reclaim:

(FPointsTo-Fractional) $\ell \overset{q_1+q_2}{\longmapsto} v \dashv\vdash \ell \overset{q_1}{\longmapsto} v * \ell \overset{q_2}{\longmapsto} v$

(FPointsTo-Agree) $\ell \overset{q_1}{\longmapsto} v_1 * \ell \overset{q_2}{\longmapsto} v_2 \vdash v_1 = v_2$

(FPointsTo-Read) $\{\ell \overset{q}{\mapsto} v\} * \ell \{v.\, \ell \overset{q}{\mapsto} v\}$

However, if the fraction is unknown, the full ownership cannot be recovered, leading to permanent loss of write and reclamation permission (hence "leaking" the memory). Despite such a big disadvantage, leaking points-to has been widely used [Iris Team 2023a] for its *duplicability* in addition to immutability and read permission:

(LPointsTo-Duplicable) $\ell \mapsto^{?} v \dashv\vdash \ell \mapsto^{?} v * \ell \mapsto^{?} v$

(LPointsTo-Agree) $\ell \mapsto^{?} v * \ell \mapsto^{?} v' \vdash v = v'$

(LPointsTo-Read) $\{\ell \mapsto^{?} v\} * \ell \{v.\, \ell \mapsto^{?} v\}$

As we will see shortly, duplicability is crucial for reasoning about safe dereference in `pop()`.

***Verification.*** We prove the safety of `pop()` as illustrated in Fig. 4. In particular, we prove the safety of its memory accesses at C4,8,9 as follows.

- C4: We open the invariant IsStack to get its contents. On the first conjunct $st \mapsto \{.\mathsf{head} = h_1\}$, we use PointsTo-Read to dereference $st$.head and get the pointer value $h_1$. Furthermore, if $h_1$ is not NULL, we use LPointsTo-Duplicable to obtain a copy of $h_1 \mapsto^{?} \{\mathtt{.data} = x_1,\ \mathtt{.next} = n_1\}$ from LinkedList for some $x_1$ and $n_1$. This is necessary for proving the safety of dereferencing $h_1$ at C8, because there is no guarantee that $h_1$ will still be present in the stack (and thus in the invariant) at that point. Finally, we close the invariant IsStack.
- C8: Since $h_1$ is not NULL, we can use the copied $h_1 \mapsto^{?} \{\mathtt{.data} = x_1,\ \mathtt{.next} = n_1\}$ to ensure that it is safe to dereference $h_1$ (LPointsTo-Read).
- C9: We open the invariant and obtain $\exists h_2.\, st \mapsto \{.\mathsf{head} = h_2\}$. Consider the case where $h_1 = h_2$ holds. We use PointsTo-CAS-Success to update the head pointer to the next node, and reestablish the invariant with the head node detached.

For `pop()` to satisfy the logical atomicity specification, its successful CAS should be free of the ABA problem (Fig. 1c). This is indeed the case because locations are not recycled for new nodes if memory is not reclaimed, and the nodes added to the stack are immutable. Formally, this argument corresponds to the use of LPointsTo-Agree at lines V9.1-9.2: the two $h_1 \mapsto^{?} \ldots$ assertions (obtained from C4 and C9) contain the same value.

```
V0   $\{\mathsf{IsStack}(st)\}$
C1   fun pop(st):
C3     loop:
V3.1     $\{\mathsf{IsStack}(st)\}$
V3.2     $\left\{\exists h_1, xs_1.\ st \mapsto \{.\mathtt{head} = h_1\} * \bigvee \begin{cases} h_1 = \mathsf{NULL} * xs_1 = [\,] \\ \exists x_1, xs_1', n_1.\ xs_1 = x_1 :: xs_1' \\ \quad * h_1 \mapsto^{?} \{.\mathtt{data} = x_1, .\mathtt{next} = n_1\} * \mathsf{LinkedList}(n_1, xs_1') \end{cases}\right\}$
V3.3     $\left\{\exists h_1, xs_1.\ st \mapsto \{.\mathtt{head} = h_1\} * \bigvee \begin{cases} h_1 = \mathsf{NULL} * xs_1 = [\,] \\ \exists x_1, xs_1', n_1.\ xs_1 = x_1 :: xs_1' \\ \quad * h_1 \mapsto^{?} \{\ldots\} * h_1 \mapsto^{?} \{\ldots\} * \mathsf{LinkedList}(n_1, xs_1') \end{cases}\right\}$
C4       h1 := (*st).head
V4.1     $\left\{\mathsf{IsStack}(st) * \bigvee \begin{cases} h_1 = \mathsf{NULL} \\ \exists x_1, n_1.\ h_1 \mapsto^{?} \{.\mathtt{data} = x_1, .\mathtt{next} = n_1\} \end{cases}\right\}$
C5       if h1 == NULL:
C7         return None
V7.1     $\{\mathsf{IsStack}(st) * \exists x_1, n_1.\ h_1 \mapsto^{?} \{.\mathtt{data} = x_1, .\mathtt{next} = n_1\}\}$
C8       x1 := (*h1).data; n1 := (*h1).next
V8.1     $\{\mathsf{IsStack}(st) * h_1 \mapsto^{?} \{.\mathtt{data} = x_1, .\mathtt{next} = n_1\}\}$
V8.2     $\left\{h_1 \mapsto^{?} \{\ldots\} * \exists h_2, xs_2.\ st \mapsto \{.\mathtt{head} = h_2\} * \bigvee \begin{cases} h_2 = \mathsf{NULL} * xs_1 = [\,] \\ \exists x_2, xs_2', n_2.\ xs_2 = x_2 :: xs_2' \\ \quad * h_2 \mapsto^{?} \{.\mathtt{data} = x_2, .\mathtt{next} = n_2\} * \mathsf{LinkedList}(n_2, xs_2') \end{cases}\right\}$
C9       if CAS(&(*st).head, h1, n1):
V9.1       $\{h_1 \mapsto^{?} \{\ldots\} * st \mapsto \{.\mathtt{head} = h_1\} * h_1 \mapsto^{?} \{.\mathtt{data} = x_2, .\mathtt{next} = n_2\} * \mathsf{LinkedList}(n_2, xs_2')\}$
V9.2       $\{x_1 = x_2 \land n_1 = n_2 \land \ldots\}$
V9.3       $\{h_1 \mapsto^{?} \{.\mathtt{data} = x_1, .\mathtt{next} = n_1\} * \mathsf{IsStack}(st)\}$
C11        return Some(x1)
```

Fig. 4. Verification of Treiber's stack without reclamation.

***Verification Challenges for Memory Reclamation.*** The above proof is fundamentally limited to concurrent data structures without reclamation due to its reliance on leaking points-to. To use LPointsTo-Duplicable at C4, the proof exploits the fact that the stack's memory blocks are never reclaimed. However, verification of concurrent data structures with reclamation requires an ability to grant access permission to protected pointers like LPointsTo-Duplicable, but only *temporarily* so that retired pointers can later be reclaimed. In §3 and §4, we will characterize such temporary grant of access permission with new abstract predicates representing partial ownership of pointers.

## 3 SPECIFICATION OF HAZARD POINTERS

We first introduce two predicates, Managed and Protected, that replace leaking points-to assertions. Fig. 5 presents their signatures and associated proof rules. For presentation purposes, we assume that the contents of memory blocks are *immutable* and lift this assumption in §5.

***Managed Pointer.*** The *managed* pointer predicate of the form $\mathsf{Managed}(\ell, v)$ represents the ownership of the pointer $\ell$ managed by hazard pointers.

Like the ordinary points-to predicates, the managed pointer assertion implies that $\ell$ is a valid pointer to a memory block containing the value $v$. Managed pointer assertions replace the leaking

**Predicates**

$$\mathsf{Managed}(\ell : Loc,\ v : Val) : iProp \qquad \mathsf{Protected}(tid : ThreadId,\ \ell : Loc,\ v : Val) : iProp$$

$$\mathsf{HPSlot}(tid : ThreadId,\ \ell : Loc) : iProp \qquad \mathsf{HPSlot}(tid) := \mathsf{HPSlot}(tid,\ \mathtt{NULL})$$

**Basic rules**

(Managed-New)
$$\ell \mapsto v \Rrightarrow\!\!\ast\ \mathsf{Managed}(\ell, v)$$

(Managed-Access)
$$\{\mathsf{Managed}(\ell, v)\} \ast\!\ell\ \{v.\ \mathsf{Managed}(\ell, v)\}$$

(HP-Retire)
$$\{\mathsf{Managed}(\ell, \_)\}\ \mathtt{retire}(\ell)\ \{\mathsf{True}\}$$

(Protect)
$$\left\langle \begin{matrix}\ell, v.\ src \mapsto \ell \ast \mathsf{Managed}(\ell, v)\\ \ast\ \mathsf{HPSlot}(tid)\end{matrix} \right\rangle \mathtt{protect}(tid, src) \left\langle \begin{matrix}\ell.\ src \mapsto \ell \ast \mathsf{Managed}(\ell, v)\\ \ast\ \mathsf{Protected}(tid, \ell, v)\end{matrix} \right\rangle$$

(Protected-Access)
$$\{\mathsf{Protected}(tid, \ell, v)\} \ast\!\ell\ \{v.\ \mathsf{Protected}(tid, \ell, v)\}$$

(Unprotect)
$$\{\mathsf{Protected}(tid, \_, \_)\}\ \mathtt{unprotect}(tid)\ \{\mathsf{HPSlot}(tid)\}$$

**Low-level rules**

(HPSlot-Set)
$$\{\mathsf{HPSlot}(tid, \_)\}\ \mathtt{protected[}tid\mathtt{]}\ :=\ \ell\ \{\mathsf{HPSlot}(tid, \ell)\}$$

(Protected-Managed-Agree)
$$\mathsf{Protected}(\_, \ell, v) \ast \mathsf{Managed}(\ell, v') \vdash v = v'$$

(HPSlot-Validate)
$$\mathsf{Managed}(\ell, v) \ast \mathsf{HPSlot}(tid, \ell) \Rrightarrow\!\!\ast\ \mathsf{Managed}(\ell, v) \ast \mathsf{Protected}(tid, \ell, v)$$

Fig. 5. A specification of hazard pointers.

points-to assertions in data structure invariants. For instance, in the invariant of Treiber's stack, the LinkedList predicate is changed as follows:

$$\mathsf{LinkedList}(h, xs) := \bigvee \begin{cases} h = \mathtt{NULL} \land xs = [] \\ \exists x, xs', n.\ \mathsf{Managed}(h, \{\mathtt{.data} = x,\ \mathtt{.next} = n\}) \ast \mathsf{LinkedList}(n, xs')\ . \end{cases}$$

Here, $\mathsf{Managed}(h, \{\mathtt{.data} = x, \mathtt{.next} = n\})$ replaces $h \overset{?}{\mapsto} \{\mathtt{.data} = x, \mathtt{.next} = n\}$ in §2.3. Similarly to leaking points-to, managed pointer predicates permit read access via Managed-Access.

A managed pointer assertion is introduced from a full points-to via Managed-New. Here, a *view shift* assertion $P \Rrightarrow\!\!\ast\ Q$ says that $P$ can be transformed into $Q$ while proving a Hoare triple:

(Hoare-VS)
$$\frac{P \Rrightarrow\!\!\ast\ P' \qquad \{P'\}\ e\ \{Q'\} \qquad Q' \Rrightarrow\!\!\ast\ Q}{\{P\}\ e\ \{Q\}}$$

For example, the proof of stack `push()` method (not shown here) converts the points-to of the newly pushed node into a managed pointer assertion with Managed-New and prepends it to LinkedList.

Unlike leaking points-to, the managed pointer assertion represents the unique permission to retire a pointer. To show the safety of retiring a pointer, one should provide its Managed to HP-Retire as a precondition. This precondition reflects the requirement that the retirer must first detach the memory block from the data structure. By detaching the block, one can take Managed out of the data structure invariant, obtaining the exclusive right to retire it. For instance, in Fig. 6, $\mathsf{Managed}(h_1, \_)$ is detached from the stack's invariant by the CAS at C9, which is then used for `retire(h)` at C10. Note that retirement consumes Managed, so multiple retirement of the same pointer is prevented (thus preventing double-free in `do_reclamation()`).

```
C1    fun pop($st$):
C3      loop:
V3.1      $\{\mathsf{HPSlot}(tid) * \mathsf{IsStack}(st)\}$
V3.2      $\left\{\mathsf{HPSlot}(tid) * \exists h_1, xs_1.\, st \mapsto \{.\mathsf{head} = h_1\} * \bigvee \begin{cases} h_1 = \mathsf{NULL} * xs_1 = [\,] \\ \exists x_1, xs_1', n_1.\, xs_1 = x_1 :: xs_1' \\ \quad * \mathsf{Managed}(h_1, \{\ldots\}) * \ldots \end{cases}\right\}$
C4        $h_1$ := protect($tid$, &(*$st$).head)
V4.1      $\left\{\exists h_1, xs_1.\, st \mapsto \{.\mathsf{head} = h_1\} * \bigvee \begin{cases} h_1 = \mathsf{NULL} * xs_1 = [\,] * \mathsf{HPSlot}(tid) \\ \exists x_1, xs_1', n_1.\, xs_1 = x_1 :: xs_1' \\ \quad * \mathsf{Managed}(h_1, \{\ldots\}) * \mathsf{Protected}(tid, h_1, \{\ldots\}) * \ldots \end{cases}\right\}$
V4.2      $\left\{\mathsf{IsStack}(st) * \bigvee \begin{cases} h_1 = \mathsf{NULL} * \mathsf{HPSlot}(tid) \\ \exists x_1, n_1.\, \mathsf{Protected}(tid, h_1, \{.\mathsf{data} = x_1, .\mathsf{next} = n_1\}) \end{cases}\right\}$
...                                  ...
V7.1      $\{\mathsf{IsStack}(st) * \exists x_1, n_1.\, \mathsf{Protected}(tid, h_1, \{.\mathsf{data} = x_1, .\mathsf{next} = n_1\})\}$
C8        $x_1$ := (*$h_1$).data; $n_1$ := (*$h_1$).next
V8.1      $\left\{\mathsf{Protected}(h_1, \{\ldots\}) * \exists h_2, xs_2.\, st \mapsto \{.\mathsf{head} = h_2\} * \bigvee \begin{cases} h_2 = \mathsf{NULL} * xs_1 = [\,] \\ \exists x_2, xs_2', n_2.\, xs_2 = x_2 :: xs_2' \\ \quad * \mathsf{Managed}(h_2, \{.\mathsf{data} = x_2, .\mathsf{next} = n_2\}) * \ldots \end{cases}\right\}$
C9        if CAS(&(*$st$).head, $h_1$, $n_1$):
V9.1        $\{\mathsf{Protected}(h_1, \{\ldots\}) * st \mapsto \{.\mathsf{head} = h_1\} * \mathsf{Managed}(h_1, \{\ldots\}) * \mathsf{LinkedList}(n_2, xs_2')\}$
V9.2        $\{\mathsf{Protected}(h_1, \{.\mathsf{data} = x_1, .\mathsf{next} = n_1\}) * \mathsf{Managed}(h_1, \{.\mathsf{data} = x_1, .\mathsf{next} = n_1\}) * \mathsf{IsStack}(st)\}$
C10         retire($h_1$); unprotect($tid$)
V10.1       $\{\mathsf{HPSlot}(tid) * \mathsf{IsStack}(st)\}$
```

Fig. 6. Verification of Treiber's stack with hazard pointers.

***Protected Pointer.*** When a thread *tid* successfully protects a pointer $\ell$, it obtains the *protected* pointer predicate of the form $\mathsf{Protected}(tid, \ell, v)$, which represents the temporary permission for *tid* to access $\ell$. Similarly to managed pointer predicate, $\mathsf{Protected}(tid, \ell, v)$ implies that $\ell$ is a valid pointer to a memory block with value $v$.

$\mathsf{Protected}(tid, \ell, v)$ can be introduced by calling the `protect()` function (Protect). It returns a protected pointer assertion for $\ell$ loaded from *src* when $\mathsf{Managed}(\ell, \_)$ is available. In other words, protection is established only when the user shows that the pointer is not retired. Protect is formulated as a logically atomic triple so that the user can access the data structure invariant (LAT-Inv-Acc) to provide $src \mapsto \ell$ and $\mathsf{Managed}(\ell, \_)$ as the precondition. It additionally takes $\mathsf{HPSlot}(tid)$, the permission for *tid* to protect a pointer, which is created when the *tid* is spawned. The `unprotect()` function eliminates the protected pointer and returns back the protection permission (Unprotect).

The protected pointer assertion replaces the leaking points-to assertion used by each thread to reason about the safety of using the pointer. In Fig. 6, the thread obtains $\mathsf{Protected}(tid, h_1, \_)$ at C4 via Protect and uses it to show the safety of dereference at C8 via Protected-Access (analogous to LPointsTo-Read). To show that the protection prevents the ABA problem in the successful CAS at C9, we use Protected-Managed-Agree (analogous to LPointsTo-Agree) to conclude that the node protected at C4 has not changed (V9.1-9.2).

For sophisticated data structures with custom validation strategies, low-level rules for writing to the protected pointer list slot and validation are needed. As shown in HPSlot-Set, $\mathsf{HPSlot}(tid, \ell)$

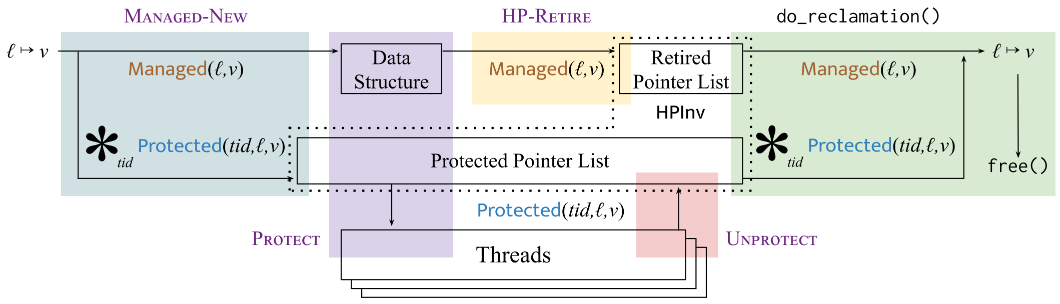


Fig. 7. The life cycle of pointer ownership in hazard pointers.

records the value written in the thread's protected pointer list slot (HPSlot($tid$) is abbreviation of HPSlot($tid$, NULL)). After $\ell$ is written to the slot, HPSlot-Validate transforms HPSlot to Protected($tid, \ell, v$) given Managed($\ell, v$). This rule is applied after running a validation check that confirms that $\ell$ is not retired. For example, Protect is proved by applying HPSlot-Set at C14 (Fig. 2) and HPSlot-Validate at C16 when the condition evaluates to true (see appendix [Jung et al. 2023]). We discuss its application to the validation strategy of the Harris-Michael list in §5.

## 4 VERIFICATION OF HAZARD POINTERS

We verify the specification of hazard pointers (§3) in concurrent separation logic. Essentially, hazard pointers is a mechanism to distribute and recollect the partial ownerships of pointers in the form of managed pointer assertion and the protected pointer assertions for each thread. Fig. 7 overviews the life cycle of a points-to assertion ($\ell \mapsto v$) transferred and shared among the memory allocator, the data structure, the protecting threads, and the reclaiming thread. In cyan area, Managed-New splits $\ell \mapsto v$ into a Managed assertion and Protected assertions for *each* thread. Then, Protected assertions are transferred to the protected pointer list of the invariant HPInv of hazard pointers (see below for details). In purple area, Protect passes the Protected assertion from the protected pointer list to the protecting thread in the presence of the Managed assertion, and in pink area, Unprotect returns the Protected assertion back. In yellow area, HP-Retire transfers the Managed assertion detached from the data structure's invariant to the retired pointer list. After all the protected pointer assertions for a retired block are returned from the threads, in green area, do_reclamation() reconstructs the points-to assertion from them and reclaims it.

### 4.1 Fractional Ownership of Pointer

To model the per-thread ownership of a pointer, we use a variant of fractional points-to assertion equipped with a permission algebra [Vafeiadis 2011]. Given a set, say $X$, the fractional points-to assertion of its powerset algebra with disjoint union, $\wp(X)_{\uplus}$, satisfies the following proof rules:

$$\ell \mapsto v \dashv\vdash \ell \xmapsto{X} v \qquad\qquad \ell \xmapsto{s_1 \uplus s_2} v \dashv\vdash \ell \xmapsto{s_1} v * \ell \xmapsto{s_2} v$$

For hazard pointers, we use the permission algebra $\wp(\mathit{ThreadId} \cup \{\star\})_{\uplus}$, where each $tid \in \mathit{ThreadId}$ represents the thread $tid$'s protected pointer and $\star$ represent the managed pointer:

$$\text{Protected}(tid, \ell, v) := \ell \xmapsto{\{tid\}} v * \ldots \qquad\qquad \text{Managed}(\ell, v) := \ell \xmapsto{\{\star\}} v * \ldots .$$

The rules Managed-Access, Protected-Access, and Protected-Managed-Agree immediately follow from the analogous rules for fractional points-to.

*Invariant*: $\exists P, D, V, R.\quad p \overset{1/2}{\longmapsto} P * d \overset{1/2}{\longmapsto} D * \gamma_{\mathrm{v}} \overset{1/2}{\Longmapsto} V * \gamma_{\mathrm{r}} \overset{1/2}{\Longmapsto} R *$
$\neg(V \wedge R) * (\neg(V \vee R) \Rightarrow (r \mapsto \mathtt{false})) * (V \Rightarrow P) * (R \Rightarrow D)$

$T_1$

V10 $\{p \overset{1/2}{\longmapsto} \mathtt{false} * \gamma_{\mathrm{v}} \overset{1/2}{\Longmapsto} \mathtt{false}\}$

C11 `*p := true`

V11 $\{p \overset{1/2}{\longmapsto} \mathtt{true} * \gamma_{\mathrm{v}} \overset{1/2}{\Longmapsto} \mathtt{false}\}$

C12 `if !*d:`

V12 $\{p \overset{1/2}{\longmapsto} \mathtt{true} * \gamma_{\mathrm{v}} \overset{1/2}{\Longmapsto} \mathtt{true} * r \mapsto \mathtt{false}\}$

C13 `  assert(!*r)`

V13 $\{\ldots\}$

C14 `*p := false`

V14 $\{p \overset{1/2}{\longmapsto} \mathtt{false} * \gamma_{\mathrm{v}} \overset{1/2}{\Longmapsto} \mathtt{false}\}$

$T_2$

V20 $\{d \overset{1/2}{\longmapsto} \mathtt{false} * \gamma_{\mathrm{r}} \overset{1/2}{\Longmapsto} \mathtt{false}\}$

C21 `*d := true`

V21 $\{d \overset{1/2}{\longmapsto} \mathtt{true} * \gamma_{\mathrm{r}} \overset{1/2}{\Longmapsto} \mathtt{false}\}$

C22 `if !*p:`

V22 $\{d \overset{1/2}{\longmapsto} \mathtt{true} * \gamma_{\mathrm{r}} \overset{1/2}{\Longmapsto} \mathtt{true} * r \mapsto \mathtt{false}\}$

C23 `  *r := true`

V23 $\{d \overset{1/2}{\longmapsto} \mathtt{true} * \gamma_{\mathrm{r}} \overset{1/2}{\Longmapsto} \mathtt{true} * r \mapsto \mathtt{true}\}$

Fig. 8. The essence of the synchronization between protection and reclamation

## 4.2 The Essence of Synchronization between Protection and Reclamation

Fig. 7 shows that there is a contention for protected pointer predicates owned by the protected pointer list between the protection of threads (Protect) and the reclamation (the `do_reclamation()` function). We reason about the synchronization between protection and reclamation as follows.

The program in Fig. 8 schematically illustrates the essence of the synchronization between the protecting thread and the reclaiming thread contending for a single memory block. (For now, please ignore the invariant and the proof lines.) The program consists of three locations $d$, $p$, and $r$ that record the status of the block: **(1)** $d$ for whether the block is *detached* from the data structure; **(2)** $p$ for whether the block is *protected*; and **(3)** $r$ for whether the block is *reclaimed*. The left thread ($T_1$) represents a thread that protects and accesses a memory block. It protects the block and validates if the block is not detached yet (Protect, corresponding to C11-12). If validated, it accesses the block (Protected-Access, corresponding to succesful assert at C13), and finally, terminates the protection (Unprotect, corresponding to C14). The right thread ($T_2$) represents a thread that retires and reclaims the memory block. It detaches the block (HP-Retire, corresponding to C21), checks if the block is not protected, and in that case, reclaims the block (the `do_reclamation()` function, corresponding to C22-23).

We can informally reason about the above program's safety by case analysis. **(1)** If C11 is executed before C21, $p$ is set before C22, so $T_2$ does not reach C23 and set $r$; and **(2)** if C21 is executed before C11, $d$ is set before C12, so $T_1$ does not reach C13 and assert that $r$ is not set. In either case, the access to the block (C13) must happen before the reclamation (C23), thereby proving the assertion.

We formalize this informal reasoning by adopting Mével et al. [2020, §5.4]'s approach for verifying Peterson's *mutual exclusion* algorithm to the context of safe memory reclamation. The key idea is to introduce *ghost locations* $\gamma_{\mathrm{v}}$ and $\gamma_{\mathrm{r}}$ that record whether protection of the block is *validated* and whether the block is *reclaimed*, and relate them to physical locations $p$ and $r$, respectively. To this end, we maintain the following invariant which we call *mutual exclusion protocol*:

- Introduce four variables: $P$ and $D$ for the value stored in physical locations $p$ and $d$; and $V$ and $R$ for the values recorded in $\gamma_{\mathrm{v}}$ and $\gamma_{\mathrm{r}}$. $V$ and $R$ are tracked using the *ghost variable assertions*

$$\mathsf{HPInv} \coloneqq \boxed{\begin{array}{l}\exists ps.\, \mathtt{protected} \overset{1/2}{\longmapsto} ps * (\text{Managed and resource of retired pointers}) * \\ \quad \mathop{\Huge *}_{p} \left( \exists D.\, \gamma_{\mathrm{d}}(\ell) \overset{1/2}{\Longmapsto} D * \mathop{\Huge *}_{tid} \left( \begin{array}{r} \exists V, R.\, \gamma_{\mathrm{v}}(\ell, tid) \overset{1/2}{\Longmapsto} V * \\ \gamma_{\mathrm{r}}(\ell, tid) \overset{1/2}{\Longmapsto} R * \ldots \end{array} \right) \right) * \ldots \end{array}}$$

$$\mathsf{HPSlot}(tid, \ell) \coloneqq \mathtt{protected}[tid] \overset{1/2}{\longmapsto} \ell * \mathop{\Huge *}_{p} \gamma_{\mathrm{v}}(\ell, tid) \overset{1/2}{\Longmapsto} \mathtt{false}$$

$$\mathsf{Protected}(tid, \ell, v) \coloneqq \ell \overset{\{tid\}}{\longmapsto} v * \mathtt{protected}[tid] \overset{1/2}{\longmapsto} \ell * \gamma_{\mathrm{v}}(\ell, tid) \overset{1/2}{\Longmapsto} \mathtt{true} * \mathop{\Huge *}_{\ell' \neq \ell} \gamma_{\mathrm{v}}(\ell, tid) \overset{1/2}{\Longmapsto} \mathtt{false}$$

$$\mathsf{Managed}(\ell, v) \coloneqq \ell \overset{\{\star\}}{\longmapsto} v * \gamma_{\mathrm{d}}(\ell) \overset{1/2}{\Longmapsto} \mathtt{false} * \mathop{\Huge *}_{tid \in \mathbb{N}} \gamma_{\mathrm{r}}(\ell, tid) \overset{1/2}{\Longmapsto} \mathtt{false}$$

Fig. 9. Definition of the predicates and invariant of hazard pointers.

which behave like (fractional) points-to assertions for ghost locations:

(Ghost-Var-Agree)
$$\gamma \overset{f}{\Longmapsto} x * \gamma \overset{f'}{\Longmapsto} x' \vdash x = x'$$

(Ghost-Var-Fract)
$$\gamma \overset{f}{\Longmapsto} x * \gamma \overset{f'}{\Longmapsto} x \dashv\vdash \gamma \overset{f+f'}{\Longmapsto} x$$

(Ghost-Var-Update)
$$\gamma \overset{1}{\Longmapsto} x \Rrightarrow\!\!* \; \gamma \overset{1}{\Longmapsto} y$$

- Distribute the ownership of $p$, $d$, $\gamma_{\mathrm{v}}$, and $\gamma_{\mathrm{r}}$ to the invariant and threads. $T_1$ can write to $p$ and $\gamma_{\mathrm{v}}$ and read from the other two by opening the invariant, and the other way around for $T_2$.
- Require the condition $\neg(V \land R)$, *i.e.*, mutual exclusion of validation and reclamation. When neither is true, the invariant keeps ownership of $r$ with the condition $\neg(V \lor R) \Rightarrow (r \mapsto \mathtt{false})$. When $T_1$ validates the protection, it knows that the block is not reclaimed thanks to mutual exclusion, hence the ownership of $r$ will move to $T_1$, and vice versa for reclamation in $T_2$.
- Require $(V \Rightarrow P)$: to validate the protection, $T_1$ should have announced the protection already; and $(R \Rightarrow D)$: to reclaim the block, $T_2$ should ensure that the block has been detached already.

Using the invariant, proof of $T_1$'s safety proceeds as follows (proof for $T_2$ is similar).

- V11: We open the invariant to get the half ownership of $p$, combine it with the other half owned by $T_1$, and write $\mathtt{true}$ to it. Then, we split it in half to close the invariant.
- V12: We consider the case where the block is not detached yet and thus the validation succeeds. We open the invariant to get the full ownership of $\gamma_{\mathrm{v}}$. Since $D = \mathtt{false}$, we derive $R = \mathtt{false}$ from $(R \Rightarrow D)$. We take $r \mapsto \mathtt{false}$ out of the invariant by setting $\gamma_{\mathrm{v}} \Mapsto \mathtt{true}$ (via Ghost-Var-Update), so that it does not have to be returned to the invariant. Since $p \overset{1/2}{\longmapsto} \mathtt{true}$, $(V \Rightarrow P)$ is maintained, and we can close the invariant.
- V12-13: We read from $r$ using $r \mapsto \mathtt{false}$. The assertion succeeds.
- V14: We open the invariant to get the full ownership of $p$ and $\gamma_{\mathrm{v}}$; write to $p$; set $\gamma_{\mathrm{v}} \Mapsto \mathtt{false}$ and return $r \mapsto \mathtt{false}$ back if necessary; and close the invariant.

## 4.3 Generalization to Multiple Pointers and Threads

Fig. 9 presents the definition of the predicates and invariant of hazard pointers supporting multiple pointers and threads. The invariant $\mathsf{HPInv}$ of hazard pointers is assumed in every proof rule presented in Fig. 5. The invariant generalizes that in Fig. 8 by collecting mutual exclusion protocols that govern ownership transfer of $\ell \overset{\{tid\}}{\longmapsto} \_$ for each pointer $\ell$ and thread $tid$. More specifically, each slot $\mathtt{protected}[tid]$ corresponds to $p$ (in Fig. 8), the per-pointer per-thread ghost locations $\gamma_{\mathrm{v}}(\ell, tid)$ and $\gamma_{\mathrm{r}}(\ell, tid)$ to $\gamma_{\mathrm{v}}$ and $\gamma_{\mathrm{r}}$, and the per-pointer ghost location $\gamma_{\mathrm{d}}(\ell)$ to $d$, respectively. $\mathsf{HPSlot}(tid, \_)$ carries $tid$'s $\gamma_{\mathrm{v}}$ flags for all possible pointers, and $\mathsf{Managed}(\ell, \_)$ asserts that $\gamma_{\mathrm{d}}(\ell)$ flag is false. The verification of the specification proceeds as follows.

**Types and predicates**

$$\mathit{BlockRes} \coloneqq \mathit{Loc} \to \mathit{Val} \to \mathit{BlockId} \to \mathit{iProp} \qquad \mathsf{Managed}(\ell : \mathit{Loc}, i : \mathit{BlockId}, P : \mathit{BlockRes}) : \mathit{iProp}$$

$$\mathsf{Protected}(\mathit{tid} : \mathit{ThreadId}, \ell : \mathit{Loc}, i : \mathit{BlockId}, P : \mathit{BlockRes}) : \mathit{iProp}$$

**Rules**

(Managed-New-Full)

$$\ell \mapsto v * (\forall i.\, i \text{ fresh} \Rrightarrow\!\!* \; P(\ell, v, i)) \Rrightarrow\!\!* \; \exists i.\, i \text{ fresh} * \mathsf{Managed}(\ell, i, P)$$

(HPSlot-Validate-Full)

$$\mathsf{Managed}(\ell, i, P) * \mathsf{HPSlot}(\mathit{tid}, \ell) \Rrightarrow\!\!* \; \mathsf{Managed}(\ell, i, P) * \mathsf{Protected}(\mathit{tid}, \ell, i, P)$$

(Protected-Access-Full)

$$\frac{\{\exists v.\, \ell \mapsto v * P(\ell, v, i)\}\; e\; \{\ell \mapsto v' * P(\ell, v', i)\} \qquad e \text{ physically atomic}}{\{\mathsf{Protected}(\mathit{tid}, \ell, i, P)\}\; e\; \{\mathsf{Protected}(\mathit{tid}, \ell, i, P)\}}$$

(Protected-Managed-Agree-Full)

$$\mathsf{Protected}(\mathit{tid}, \ell, i, P) * \mathsf{Managed}(\ell, i', P) \vdash i = i'$$

Fig. 10. Excerpt from the full specification of hazard pointers.

- HPSlot-Validate: When $\mathit{tid}$ validates protection of $\ell$, $\gamma_{\mathrm{v}}(\ell, \mathit{tid})$ is set and the partial ownership of $\ell$ is granted to $\mathsf{Protected}(\mathit{tid}, \ell, \_)$.
- HP-Retire: $\gamma_{\mathrm{d}}(\ell)$ flag is set, and its resources are temporarily stored in HPInv.
- Safety of `do_reclamation()`: It picks up the resources of retired pointers and collects each thread's ownership by scanning the protected pointer list. If a retired pointer is not in the protected pointer list, it is guaranteed that `do_reclamation()` has collected the full ownership of the pointer, thus it is safe to reclaim it.

## 5 REASONING ABOUT MUTABLE MEMORY BLOCKS

The specifications from §3 are only applicable to simple data structures like Treiber's stack in which the contents of memory blocks do not change. In this section, we generalize the specification to enable verification of sophisticated data structures with mutable memory blocks. The key idea is replacing read-only fractional points-to assertions in the Managed and Protected predicates with a per-block invariant that governs the evolution of the contents of each block. Fig. 10 shows the updated signatures of each predicate and the new proof rules.

***Block Resource and ID.*** To represent per-block invariants, we introduce *block resource predicate* and parameterize Managed and Protected predicates with it, replacing the fixed value parameter. A block resource predicate of the form $P(\ell, v, i)$ depends not only on the block's address $\ell$ and contents $v$, but also on its *block ID* $i$. Block IDs are abstract values that uniquely identify different allocations of blocks. That is, if a memory block is reclaimed and reallocated, it is assigned a different block ID. Managed and Protected are also parameterized by the current block ID in order to relate each block's resource and data structure's global invariant. In verification ignoring reclamation, such relation typically is expressed only with physical pointer values, exploiting the fact that the pointer values are unique since they are not reused.

***Proof Rules.*** When registering a managed pointer with block resource predicate $P$ (Managed-New-Full), the user should additionally show that $P(\ell, v, i)$ can be created given a globally fresh $i$. During validation (HPSlot-Validate-Full), the knowledge about the block ID and resource are transferred from managed pointer to the new protected pointer predicate. Once validated, the user can temporarily access the memory block's full points-to assertion (on the current value) as well as its block resource via Protected-Access-Full while executing an atomic instruction, in a style similar to the usual invariant access rule Inv-Acc.[7] Finally, Protected-Managed-Agree-Full says that managed and protected pointer assertions of the same pointer agree on their block ID, hence protected pointers are free from the ABA problem. Intuitively, the rule holds because the presence of a protected pointer guarantees that the pointer cannot be reclaimed and reallocated, and the block ID of a block does not change as long as they are not reallocated.

***Application to the Harris-Michael List.*** Using the above specification, we have formally verified the Harris-Michael list with manual reclamation for the first time. Here, we sketch the verification of its validation method discussed in §2.1. We use the following block resource and global invariant:

$$\mathsf{HMLBlock}(\ell, v, i) \coloneqq \ldots * \bigvee \begin{cases} i \overset{1/2}{\Longmapsto} (v.\mathtt{next}, \mathsf{Unmarked}) * \mathsf{LSB}(v.\mathtt{next}) = 0 \\ i \Mapsto^{?} (v.\mathtt{next}, \mathsf{Marked}) * \mathsf{LSB}(v.\mathtt{next}) = 1 \end{cases}$$

$$\mathsf{IsHML} \coloneqq \boxed{\begin{array}{l} \exists A, L. \left( \mathop{\Large *}\limits_{i \mapsto (\ell, v) \in A} \bigvee \begin{cases} i \overset{1/2}{\Longmapsto} (v.\mathtt{next}, \mathsf{Unmarked}) * (i, \ell) \in L * (\_, v.\mathtt{next}) \in L * \cdots \\ i \Mapsto^{?} (v.\mathtt{next}, \mathsf{Marked}) * \cdots \end{cases} \right) * \\ \qquad \left( \mathop{\Large *}\limits_{(i,\ell) \in L} \mathsf{Managed}(\ell, i, \mathsf{HMLBlock}) \right) * \ldots \end{array}}$$

In HMLBlock, we use the block ID $i$ as the name for a ghost variable[8] recording the state of the `next` field. The `next` field is either not marked yet or marked permanently. The invariant IsHML holds the state of all nodes that have been added to the list ($A$), and Managed of blocks that are not detached ($L$). Specifically, IsHML maintains that the target block of an unmarked link is not detached ($(\_, v.\mathtt{next}) \in L$).

In the validation stage of the Harris-Michael list, we use Protected-Access-Full to access the block resource of the current node; if its `next` field is unmarked, open IsHML to learn that the next node is not detached; and find its Managed to validate the protection of next node with HPSlot-Validate-Full.

***Verification of the Specification.*** To accommodate per-block invariants, we replace fractional points-to assertions with fractionally owned ghost mapping from address to block ID, and put the full points-to into a per-pointer *cancellable invariant* [Jung et al. 2018], which is also fractionally ownable. If one collects full ownership of a cancellable invariant, one can deactivate it and reclaim its content. We change HPInv accordingly to govern the ownership transfer of ghost mapping and cancellable invariants.

## 6 SPECIFICATION OF RCU

We present two modular specifications of RCU: a general specification that captures the protection of critical sections (§6.1), and a refined specification tailored towards optimistic traversal (§6.2). Using these specifications, we have formally verified Harris's list with RCU for the first time. We discuss the implementation and verification of RCU in the appendix [Jung et al. 2023].

**Types and predicates**

$$\mathsf{Guard}(tid : \mathit{ThreadId},\ R : \wp(\mathit{BlockId})) : \mathit{iProp} \qquad \mathsf{RCUState}(I : \mathit{BlockId} \xrightarrow{\text{fin}} \mathit{Loc} \times \mathit{BlockStatus}) : \mathit{iProp}$$

$$\mathit{BlockStatus} := \mathsf{Active} \mid \mathsf{Retired} \mid \ldots \qquad \mathsf{BlockInfo}(i : \mathit{BlockId},\ \ell : \mathit{Loc},\ P : \mathit{BlockRes}) : \mathit{iProp}$$

**Rules**

(RCU-Lock)
$$\langle I.\, \mathsf{RCUState}(I) * \mathsf{RCUSlot}(tid)\rangle\ \mathtt{rcu\_lock}(tid)\ \langle \mathsf{RCUState}(I) * \mathsf{Guard}(tid, \{i \in \mathit{BlockId} \mid I[i] = \mathsf{Retired}\})\rangle$$

(RCU-Unlock)
$$\{\mathsf{Guard}(tid, \_)\}\ \mathtt{rcu\_unlock}(tid)\ \{\mathsf{RCUSlot}(tid)\}$$

(Managed-Protected)
$$\mathsf{Guard}(\_, R) * \mathsf{Managed}(\ell, i, P) \vdash i \notin R$$

(RCU-Retire)
$$\langle I.\, \mathsf{RCUState}(I) * \mathsf{Managed}(\ell, i, \_)\rangle\ \mathtt{retire}(\ell)\ \langle \mathsf{RCUState}(I[i \mapsto (\ell, \mathsf{Retired})])\rangle$$

(Managed-BlockInfo)
$$\mathsf{Managed}(\ell, i, P) \vdash \square\, \mathsf{BlockInfo}(i, \ell, P)$$

(Guard-Managed-Agree)
$$\frac{i \notin R}{\mathsf{BlockInfo}(i, \ell, P) * \mathsf{Guard}(\_, R) * \mathsf{Managed}(\ell, i', P) \vdash i = i'}$$

(Guard-Access)
$$\frac{\{\exists v.\, \ell \mapsto v * P(\ell, v, i)\}\ e\ \{\ell \mapsto v' * P(\ell, v', i)\} \qquad i \notin R \qquad e \text{ physically atomic}}{\mathsf{BlockInfo}(i, \ell, P) \vdash \{\mathsf{Guard}(tid, R)\}\ e\ \{\mathsf{Guard}(tid, R)\}}$$

Fig. 11. RCU base specification.

## 6.1 General Specification Characterizing Critical Sections

RCU differs from hazard pointers in that it protects *all* accesses inside a critical section. More precisely, access to a pointer is protected throughout a critical section if its retirement does not happen before the beginning of the critical section [McKenney et al. 2023]. We encode such critical section-based protection in the *guard* predicate of the form $\mathsf{Guard}(tid, R)$ presented in Fig. 11.

The guard predicate first represents the fact that thread $tid$ is in a critical section: it is introduced by RCU-Lock and eliminated by RCU-Unlock. In doing so, it is exchanged with $\mathsf{RCUSlot}(tid)$, which is the $tid$'s permission to enter a critical section. When verifying Treiber's stack with RCU (omitted), `pop()` would own $\mathsf{Guard}(tid, \_)$ from C2 to C6 or C10 in Fig. 1a.

The guard predicate also records the set, $R$, of pointers that had been already retired when the critical section began. For example, Managed-Protected says that if a block has not been retired yet (shown by Managed that serves the same purposes as for hazard pointers), then it was not retired also when the critical section began. In our stack verification, if `pop()` loads a non-null head pointer `h` at C4 in Fig. 1a, we use $\mathsf{Managed}(\mathtt{h}, \ldots)$ from the invariant to show that `h` is not in $R$.

To more precisely track $R$, we introduce the *RCU state* predicate of the form $\mathsf{RCUState}(I)$. The parameter $I$ in $\mathsf{RCUState}(I)$ describes the current status of all memory block that has been managed by RCU. For example, RCU-Retire marks the block as retired in $I$ (other states are omitted). When a guard is created by RCU-Lock for thread $tid$ from $\mathsf{RCUState}(I)$, it records the set of retired pointers as the parameter $R$. The RCU-Lock and RCU-Retire rules are formulated as logically atomic triples so that the user can build a sophisticated invariant that involves the RCU's state. In §6.2, we will sketch such an invariant to reason about optimistic traversal (§2.2).

[7]The logically atomic access rule corresponding to LAT-Inv-Acc is omitted.
[8]This is possible because block IDs are globally unique.

The guard predicate protects *all* blocks but the ones in $R$. To give a logical meaning to protection, we introduce the *block information* predicate of the form $\mathsf{BlockInfo}(i, \ell, P)$. Intuitively, $\mathsf{BlockInfo}(i, \ell, P)$ is the knowledge extracted from $\mathsf{Managed}(\ell, i, P)$ (Managed-BlockInfo) that $i$ is associated with the physical address $\ell$ and governed by the block resource $P$.[9] Given $\mathsf{BlockInfo}(i, \ell, R)$ and the fact that $i$ is protected by the guard (*i.e.*, it already has access permission for $i$), Guard-Access grants temporary access to the block's points-to assertion and block resource.[10] In addition, a pointer value associated with a protected block ID is free from the ABA problem (Guard-Managed-Agree). To justify the above proof rules, the guard predicate takes the *tid*'s fractional access permissions of all blocks but the ones in $R$ from RCU's internal invariant (omitted). In our stack verification, BlockInfo is extracted from Managed at C4 and used to dereference `h` at C8 to show that a successful `pop()` of `h` is linearizable.

## 6.2 Traversal-Friendly Specification

***Motivation.*** We are now able to show the safety of optimistic traversal in Fig. 3. The crux of the proof lies in showing that $\ell_2$ was not retired before the beginning of the critical section. At the high level, we prove this by maintaining the *traversal loop invariant*: for all $\ell_{\text{from}}$ and $\ell_{\text{to}}$, if $\ell_{\text{from}}$ currently points to $\ell_{\text{to}}$ and $\ell_{\text{from}}$ was not detached before the beginning of a critical section, then $\ell_{\text{to}}$ also was not detached before the beginning of the critical section. This invariant implies that $\ell_2$ was not retired before that point, since its premise is true when the traversal starts from the root to the first block, and blocks are detached before retirement. The loop invariant follows from the following three properties about the links among memory blocks: **(L1)** by definition, non-detached blocks can only point to non-detached blocks; **(L2)** data structures maintain invariants that a block is detached only once; and **(L3)** a newly created link does not point to detached blocks (even from already detached blocks).

However, such a proof quickly becomes complex for realistic concurrent data structures with optimistic traversal. The proof requires a user-level invariant that encodes link properties and relates the link topology with $\mathsf{RCUState}(I)$ (*i.e.*, only detached blocks are retired). In addition, we need to maintain the history of the link topology with which we assert that, when $\mathsf{Guard}(\_, R)$ was created, all non-detached pointers at that moment in history must not be in $R$. Such an invariant is quite sophisticated, *e.g.*, for Harris's list [Harris 2001] where links are concurrently updated and a chain of nodes can be detached at once. Even worse, we would have to repeat this kind of reasoning for each data structure to apply RCU.

***Design.*** To streamline such proofs, we encapsulate the above complexities in a refined and yet general specification presented in Fig. 12. The specification is built on top of our base specification (§6.1) to directly capture the traversal loop invariant as follows.

We first strengthen the meaning of the guard predicate: $\mathsf{Guard}(\_, D)$ now records the set $D$ of pointers that are known to have not been *detached* before the guard was created.

We then introduce the *block points-to* predicate of the form $\mathsf{BlockPointsTo}(\ell, i, t)$ to represent the ownership of the fragment of the link topology. $\mathsf{BlockPointsTo}(\ell, i, t)$ says that the block $i$ currently points to another block (if any) and records its information as $t$.[11] The new rule Guard-Protect-BlockPointsTo reflects the intuition that it is safe to traverse the data structure by following the links. Specifically, if $i_1$ is protected by the guard, then its current next node $i_2$ is also protected. A

[9]$\mathsf{BlockInfo}(i, \ell, P)$ is persistent (indicated by the *persistence* modality $\square$), because $i$ is not reused for another location. Therefore, the extraction does not consume Managed.

[10]The rule for logically atomic $e$ is omitted.

[11]For concise presentation, the version presented here only supports singly-linked data structures. Our formalized development lifts this assumption by having BlockPointsTo for each field of the memory block.

**Predicates**

$$\mathsf{BlockPointsTo}(\ell : \mathit{Loc},\ i : \mathit{BlockId},\ t : \mathit{Option}(\mathit{Loc} \times \mathit{BlockId} \times \mathit{BlockRes})) : \mathit{iProp}$$

$$\mathsf{Managed}(\ell : \mathit{Loc},\ i : \mathit{BlockId},\ P : \mathit{BlockRes},\ B : \wp^{+}(\mathit{BlockId})) : \mathit{iProp}$$

$$\mathsf{Detached}(\ell : \mathit{Loc},\ i : \mathit{BlockId},\ P : \mathit{BlockRes}) : \mathit{iProp}$$

**Rules**

(Guard-Protect-BlockPointsTo)

$$\frac{i_1 \notin D}{\mathsf{BlockPointsTo}(\ell_1, i_1, \mathsf{Some}(\ell_2, i_2, P2)) * \mathsf{Guard}(tid, D) \vdash i_2 \notin D}$$

(BlockPointsTo-Update)

$$\mathsf{BlockPointsTo}(\ell_1, i_1, \mathsf{Some}(\ell_2, i_2, P2)) * \mathsf{Managed}(\ell_2, i_2, P2, B_2) * \mathsf{Managed}(\ell_3, i_3, P3, B_3)$$
$$\Rrightarrow\!\!* \ \mathsf{BlockPointsTo}(\ell_1, i_1, \mathsf{Some}(\ell_3, i_3, P3)) * \mathsf{Managed}(\ell_2, i_2, P2, B_2 \setminus \{i_1\}) * \mathsf{Managed}(\ell_3, i_3, P3, B_3 \uplus \{i_1\})$$

(Managed-Detach)

$$\mathsf{Managed}(\ell, i, P, \emptyset) \Rrightarrow\!\!* \ \mathsf{Detached}(\ell, i, P)$$

(Detached-Retire)

$$\{\mathsf{Detached}(\ell, \_, \_)\}\ \mathtt{retire}(\ell)\ \{\mathsf{True}\}$$

Fig. 12. RCU traversal specification.

block points-to assertion is introduced when registering a managed pointer (rule omitted) and put into the block resource to associate the logical link structure with physical points-to assertions. For example, the block resource for Harris's list is defined as follows:

$$\mathsf{HLBlock}(\ell, v, i) := \exists t.\ \mathsf{BlockPointsTo}(\ell, i, t) * \bigvee \begin{cases} v\mathtt{.next} = \mathsf{NULL} * t = \mathsf{None} * \ldots \\ \exists i'.\, t = \mathsf{Some}(v\mathtt{.next}, i', \mathsf{HListBlock}) * \ldots \end{cases}.$$

Roughly speaking, if the node's next pointer value $v$`.next` is non-null, the block resource asserts that it is possible to traverse to its next node, which is also governed by the same block resource.

To encapsulate the reasoning about link structure properties **(L1-3)**, we add a new parameter $B$ to Managed and introduce the *detached pointer* predicate of the form $\mathsf{Detached}(\ell, i, P)$. $\mathsf{Managed}(\_, i, \_, B)$ means the block $i$ is currently *pointed by* the blocks in the multiset $B$ [Madiot and Pottier 2022], and $\mathsf{Detached}(\ell, i, P)$ is a variant of Managed that has been marked detached. To maintain **(L1)**, Managed-Detach can mark a block detached only when no other blocks point to it.[12] To maintain **(L2)**, Managed-Detach is irreversible. To maintain **(L3)**, BlockPointsTo-Update updates the target of block points-to assertion only if the new target is not yet detached. Finally, to ensure that only detached nodes can be retired, RCU-Retire is replaced with Detached-Retire.

## 7 EVALUATION

We demonstrate that our specifications of hazard pointers and RCU indeed facilitate modular verification of safe memory reclamation in the three aspects discussed in §1.2.

***Compositional Verification.*** We verified implementations of hazard pointers based on that of Meta's Folly [Meta 2023] and epoch-based RCU based on the algorithm by Parkinson et al. [2017] w.r.t. their specifications (§5 and §6) that everyone can use (without understanding their implementation) to verify concurrent data structures with reclamation.

[12]This rule is not applicable to cycles. Cycles can be retired despite that each node pointed by another node, as long as the cycle as a whole is detached. This can be supported by adapting the "cloud" assertion by Madiot and Pottier [2022].

Table 1. Quantitative analysis of the overhead of adding reclamation to the code and proof. "NR", "HP", "RCU": versions without reclamation, with hazard pointers, and with RCU, respectively. "N/A": Harris's list is not supported by hazard pointers. Lines of code and proof with reclamation are annotated with the percentage of overhead over those without reclamation in parentheses. "Total" for "NR": excluding and including Harris's list for comparison with HP and RCU, respectively.

| Data Structure | NR Code | HP Code | RCU Code | NR Proof | HP Proof | RCU Proof |
|---|---|---|---|---|---|---|
| Counter | 23 | 30 (+30.4%) | 30 (+30.4%) | 140 | 175 (+25.0%) | 168 (+20.0%) |
| Treiber's Stack [Treiber 1986] | 38 | 52 (+36.8%) | 51 (+34.2%) | 199 | 248 (+24.6%) | 233 (+17.1%) |
| Elimination Stack [Hendler et al. 2004] | 54 | 71 (+31.5%) | 70 (+29.6%) | 297 | 404 (+36.0%) | 384 (+29.3%) |
| Michael-Scott Queue [Michael and Scott 1996] | 55 | 76 (+38.2%) | 68 (+23.6%) | 464 | 620 (+33.6%) | 578 (+24.6%) |
| DGLM Queue [Doherty et al. 2004] | 55 | 76 (+38.2%) | 68 (+23.6%) | 463 | 775 (+67.4%) | 731 (+57.9%) |
| Harris's List [Harris 2001] | 113 | N/A | 144 (+27.4%) | 1,389 | N/A | 1,805 (+29.9%) |
| Harris-Michael List [Michael 2002] | 96 | 146 (+52.1%) | 119 (+24.0%) | 1,171 | 1,278 (+ 9.1%) | 1,473 (+25.8%) |
| Chase-Lev Deque [Chase and Lev 2005] | 82 | 90 (+ 9.8%) | 89 (+ 8.5%) | 1,113 | 1,293 (+16.2%) | 1,284 (+15.4%) |
| RDCSS [Harris et al. 2002] | 52 | 75 (+44.2%) | 68 (+30.8%) | 400 | 530 (+32.5%) | 467 (+16.8%) |
| Total | 455/568 | 616 (+35.4%) | 707 (+24.5%) | 4,247/5,636 | 5,323 (+25.3%) | 7,123 (+26.4%) |

***General Applicability.*** We verified 9 concurrent data structures, listed in Table 1, using our specifications of hazard pointers and RCU. To the best of our knowledge, we are the first to formally verify strong specifications of Harris's list, the Harris-Michael list, the Chase-Lev deque, and RDCSS with manual memory reclamation. This selection of data structures showcases the wide applicability of our specifications to the following tricky features: **(1)** mutable memory blocks (§5): all except counter and Treiber's stack; **(2)** multiple block resources: elimination stack (value and offer); **(3)** complex validation in hazard pointers (§2.1): Michael-Scott queue, Harris-Michael list, RDCSS; **(4)** complex invariant for safe retirement (see below): Michael-Scott queue and DGLM queue; **(5)** *prophecy variable* [Jung et al. 2019]: Harris's list, Harris-Michael list, RDCSS; **(6)** optimistic traversal (§2.2): Harris's list; and **(7)** dynamically-sized blocks: Chase-Lev deque.

***Easy Integration.*** We argue that our specifications streamline the additional reasoning for introducing memory reclamation to existing concurrent data structures without reclamation. To this end, we compare the lines of Coq code for implementation and proof for logical atomicity (§2.3) of concurrent data structures with and without reclamation. The result is summarized in Table 1. The total code overhead of applying hazard pointers and RCU is 35.4% and 24.5%, respectively. Hazard pointers generally incurs higher overhead than RCU because it requires additional code for protection, especially for data structures with complex validation such as the Harris-Michael list. The total proof overhead of applying hazard pointers and RCU is 25.3% and 26.4%, respectively. The total proof overhead for hazard pointers is smaller because of the outlier, Harris-Michael list.

Overall, the proof overhead is on par with the code overhead across the 9 concurrent data structures. Some data structures exhibit moderately higher proof overhead than code overhead for the following reasons. **(1)** Elimination stack with reclamation requires more precise tracking of ownership than that without reclamation. When ignoring reclamation, the invariant needs to track only the most recent offer. On the other hand, with reclamation, the invariant needs to track all previous offers to prove the safety of retiring an offer, which may have been overridden by others. **(2)** Michael-Scott queue and DGLM queue with reclamation require capturing additional invariants on their head and tail indexes for safe retirement of nodes. For the former, the head index should not "overtake" the tail index to ensure every unlinked node is unreachable from the tail; and the latter features an optimization that requires more complex invariant on indexes.

It is worth noting that the additional proofs for hazard pointers and RCU resemble each other: switching the pointer predicates and proof rules between them *almost* works. The only exception is the validation in hazard pointers, which requires completely new proofs. Interestingly, the Harris-Michael list with hazard pointers exhibits significantly smaller proof overhead because validation simplifies the reasoning related to prophecy variables.

## 8 RELATED WORK

***Program Logic for SMR Schemes.*** Various program logic-based approaches have been proposed to verify concurrent data structures with manual memory management. However, none of them support all the three criteria of modular verification discussed in §1.2 at the same time.

Parkinson et al. [2007] verified the safety of Treiber's stack with hazard pointers in concurrent separation logic using ghost variables for the status of protection and reclamation. Their verification is *not generally applicable* because it makes a simplifying assumption that the stack's memory blocks are never reclaimed. Therefore, their verification result only applies to specialized use cases such as using the stack as a component in the memory allocator.

Tofan et al. [2011] verified linearizability and lock freedom of Treiber's stack and Michael-Scott queue with hazard pointers in temporal logic. Their verification is *not compositional* because it crucially relies on a relational invariant over the states of concurrent data structures and SMR schemes. For instance, their key invariant, ishazard, depends on the implementation of both the `pop()` function of Treiber's stack and the `scan()` function of hazard pointers.

Fu et al. [2010] verified the safety of Treiber's stack with hazard pointers in temporal separation logic supporting rely-guarantee reasoning. They construct an invariant on the history of execution traces using temporal logic connectives. Their method is *not compositional* because their invariants and rely-guarantee conditions tightly couple the implementation details of stack and hazard pointers. In addition, their method is *not generally applicable* because they target a blocking implementation of `retire()` and exploit this fact in their proof to simplify the invariants.

Gotsman et al. [2013] presented a principled approach for applying temporal separation logic to SMR schemes. As discussed in §1.2, while their approach is elegant, it is *not compositional* because it exposes the implementation details of SMR schemes. For instance, their verification of an RCU-based counter [Gotsman et al. 2013, §5] maintains the following invariant:

$$\Upsilon_{\mathsf{RCU}} \coloneqq \forall \ell, tid.\ \Big(S(tid, 1) \text{ since } \boxed{C \mapsto \ell * \ell \mapsto \_}\Big) \implies \boxed{\ell \mapsto_{\mathsf{e}} \_}\ ,$$

where $S(tid, 1)$ means that $tid$ is in an RCU critical section and $C \mapsto \ell * \ell \mapsto \_$ means that the memory block is reachable from the counter. In the verification of the RCU-based counter, one has to show that $\Upsilon_{\mathsf{RCU}}$ is *stable* under RCU's actions. This means that the RCU-based counter's proof must know the details of RCU's action and the definition of the $S(tid, 1)$ predicate. The client of the RCU-based counter has a similar issue. When the clients of the counter set up their own rely-guarantee conditions, they should prove that each condition preserves $\Upsilon_{\mathsf{RCU}}$, so the clients also need to reason about the implementation details of both the counter and RCU. It is unclear how to modularize the proofs conducted with their approach since this invariant inherently intertwines concepts of SMR schemes ("protected") and concurrent data structures ("reachable"). In contrast, our modular specification of SMR schemes decouples the two concepts with carefully designed abstract predicates for protected pointers and others.

In addition, it is unclear whether their approach can be *easily integrated*, as it was evaluated only for simple data structures without mutable memory blocks (§5 and §7).

Tassarotti et al. [2015] accounted for RCU's synchronization purely in terms of ownership transfer in a separation logic for relaxed memory [Turon et al. 2014]. A simplified version of their verification is mechanized in the Iris separation logic framework [Iris Team 2023b; Jung et al. 2018, 2015] by Kaiser et al. [2017]. However, as discussed in §1.2, their approach is *not compositional*, because they monolithically verified a linked list integrated with an implementation of RCU.

Essentially, our contributions are modularization and generalization of their verification. Our base specification for RCU (§6.1) abstracts their reasoning about the transfer of partial ownership of pointers for each thread, and the traversal-friendly specification (§6.2) encapsulates their reasoning about the history of links among memory blocks and generalizes it to multiple writers.

Furthermore, we additionally take account of the following features of general-purpose RCU.

- *Temporary Deactivation*: Their RCU does not support temporary deactivation of critical section and requires each thread to periodically refresh the critical section to guarantee the progress of reclamation. This means that all pointers are protected at the start. Therefore, in their proof, the ownership flows only from the readers to the writer. In contrast, verifying the general-purpose RCU requires reasoning about bidirectional ownership transfer in `rcu_lock()` and `rcu_unlock()`. To verify it, we use a variant of mutual exclusion protocol presented in §4.2.
- *Non-blocking Reclamation*: In their RCU, a designated reclaimer blocks until all retired pointers become safe to reclaim. In contrast, the RCU we verified allows multiple threads to concurrently and selectively reclaim safe-to-reclaim pointers without blocking. To verify it, we reason about the reclaimability of each retired pointer individually.

These limitations are in part due to the complexity of relaxed memory models. Especially, deactivation requires SC fence (*e.g.*, `atomic_thread_fence(memory_order_seq_cst)` in C/C++) in relaxed memory model, which is not supported by the logic [Turon et al. 2014] they used.

***Automated Verification Tools for SMR Schemes.*** Meyer and Wolff [2019a,b]; Wolff [2021] designed an automated linearizability checker, as discussed in §1.2. Given the result that a data structure without reclamation is linearizable (checked by the Cave verifier [Vafeiadis 2010a,b]), their verifier additionally checks the following, which as a whole implies linearizability of the data structure integrated with an SMR scheme: **(1)** the SMR implementation satisfies the SMR specification, which is an automaton that over-approximates the set of pointers that may be freed; and **(2)** the data structure with SMR scheme does not suffer from the ABA problem. The task **(2)** is further broken down into two verification tasks: **(2-1)** type-checking the data structure code annotated with invariants and transformed by applying atomicity abstraction; and **(2-2)** checking the invariant annotations using Cave.

While their approach is conceptually compositional and easily integrated, it is *not generally applicable* in practice because of the complexity of the check **(2-2)**. First, it suffers from the unsoundness of the backend verifier Cave. This resulted in failed verification in the DGLM queue, the Harris-Michael list, and Harris's list [Wolff 2021, §8.8]. Second, considering that this check takes much more time than the original verification task (linearizability under no reclamation) for complex data structures, it is likely to time out even if they did not have any unsoundness issues. Although this problem can be resolved by applying a stronger backend verifier such as Plankton [Meyer et al. 2022], it is unclear whether it can handle complex reasoning required in Harris's list for optimistic traversal and retirement of a chain of detached nodes, which involve complex shape invariants.

In addition, it is worth noting that their tool is specialized for verification of linearizability. While linearizability has been considered the de facto standard for concurrent data structure specification, it is difficult for clients to use linearizability for modular program verification: linearizability is defined outside program logics so that it is not able to express rely-guarantee conditions between the library and client [da Rocha Pinto et al. 2014]; it does not support ownership transfer [Gotsman and Yang 2012; Jacobs and Piessens 2011]; and it is not applicable to highly concurrent libraries with weaker guarantees [Afek et al. 2010; Derrick et al. 2014; Haas et al. 2016; Henzinger et al. 2013; Jagadeesan and Riely 2014]. On the other hand, we verified logically atomic triples, which can encode not only linearizability but also other correctness conditions [Dang et al. 2022].

Alglave et al. [2018] proposed a specification of RCU and proved the specification for an implementation of RCU in the Linux kernel's relaxed memory model. They also verify small client programs using RCU by model checking. However, they do not verify concurrent data structures, and it is unclear whether their model checking-based verification scales well to more complex concurrent data structures and larger programs consisting of multiple data structures.

Kuru and Gordon [2019] proposed a specification of RCU that guarantees memory safety and the absence of memory leaks. Their specification is formulated as a type system whose derivation essentially envelopes a separation logic proof. However, their specification is not validated against an implementation of RCU and is limited to single-writer and tree-shaped data structures.

***Verification of Other Memory Management Methods.*** Dang et al. [2019]; Doko and Vafeiadis [2017] verified an implementation of the *atomic reference counter* (ARC) under a relaxed memory model. ARC is simpler than the other SMR schemes because synchronization is centralized to the counter variables. In contrast, the synchronization of hazard pointers and epoch-based RCU is decentralized, *e.g.*, to the retired and protected pointer list.

Doherty et al. [2004]; Krishna et al. [2017] verified concurrent data structures that use *free list*, which is a memory recycling mechanism that keeps retired memory blocks in a list instead of returning the memory to the allocator. Since it is trivial to guarantee the safety of dereferencing pointers managed by a free list, they essentially do not reason about the safety of reclamation.

Madiot and Pottier [2022] designed a separation logic for reasoning about memory usage in a garbage collected language. Specifically, they reason about *logically deallocated* memory blocks, *i.e.*, blocks that are unreachable and thus can be reclaimed by GC. To this end, the logic uses *pointed-by* assertion of form $\ell \hookleftarrow L$, which tracks the multiset $L$ of immediate predecessor blocks of block $\ell$. The design of our traversal-aware specification for RCU (§6.2) adapts this interface to reason about detached blocks. The notable difference is that our logic tracks the history of links, while their logic only tracks the current state of links. This is necessary to support Guard-Protect-BlockPointsTo, which talks about the link status at some moment in the past when the critical section started.

## 9 FUTURE WORK

***Application to Other SMR Schemes.*** We conjecture that our style of verification generalizes to many state-of-the-art SMR schemes [Alistarh et al. 2017, 2018; Brown 2015; Kang and Jung 2020; Nikolaev and Ravindran 2020, 2021; Sheffi et al. 2021; Singh et al. 2021; Wen et al. 2018], since they are essentially hybrids of hazard pointer and RCU. As a preliminary evaluation, we have sketched a specification that commonly characterizes DEBRA+, PEBR, and NBR in the appendix [Jung et al. 2023].

***Proof Automation.*** We conjecture that the additional proof required for the usage of our SMR scheme specifications can be largely automated using Diaframe [Mulder and Krebbers 2023; Mulder et al. 2022], a proof automation framework for Iris. Since our specifications follow Iris's convention, it would be straightforward to design automation hints. As a preliminary evaluation, we wrote such hints for our hazard pointer and RCU specification and *automatically proved* Treiber's stack.

***Relaxed Memory Model.*** In this work, we have assumed the sequentially consistent memory model. As future work, we will adapt our verification to the iRC11 [Dang et al. 2019], a separation logic for C/C++'s relaxed memory model. We expect to encounter two technical challenges. First, the specification should be based on partial orders among events. For example, our RCU base specification (§6.1) should be modified to track memory blocks whose retirement does not *happen-before* at each moment. To this end, we will make use of the specification of RCU by Alglave et al. [2018] and the specification methodology by Dang et al. [2022]. Second, we need logic for SC fences, which are necessary for the implementation of general-purpose SMR schemes (discussed above). While an SC fence can be modeled as a combination of release/acquire fences and an atomic read-modify-write to a ghost location, which are already supported by existing logics [Vafeiadis 2017], this approach has not been applied to a substantial case study.

## ACKNOWLEDGMENTS

We thank the OOPSLA'23 reviewers for valuable feedback and suggestions for improvements. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT2201-06.

## DATA AVAILABILITY STATEMENT

The Coq development and appendix for this paper can be found in [Jung et al. 2023].

## A PROOF OF HAZARD POINTER PROTECT RULE

We discuss the proof of the logically atomic PROTECT rule.

***Specification.*** We first need to fix the rule as follows:

(PROTECT-FIXED)

$$\forall src, tid, \ell_0.\, \mathsf{HPSlot}(tid, \ell_0) \mathrel{-\!\!*} \langle \ell, v.\, src \mapsto \ell * \mathsf{Managed}(\ell, v)\rangle\, \texttt{protect}(tid, src)\, \langle \ell.\, src \mapsto \ell * \mathsf{Managed}(\ell, v) * \mathsf{Protected}(tid, \ell, v)\rangle$$

There are three modifications. First, HPSlot in the precondition may contain any value $\ell_0$, because `protect()` does not require the protected pointer slot's value to be `NULL`. Second, for clarification, variables *src*, *tid*, and $\ell_0$ are explicitly universally quantified. Lastly, HPSlot is moved out from the triple's precondition to the left side of a wand ($\mathrel{-\!\!*}$). This means that the triple can be used only when $\mathsf{HPSlot}(tid, \ell_0)$ is provided upfront. To understand why the last change is necessary, we need a deeper understanding of logically atomic Hoare triples.

An ordinary Hoare triple $\{P(x)\}\, e\, \{v.\, Q(x, v)\}$ asserts that $e$ is given the ownership of $P(x)$ at the start of its execution and returns the ownership of $Q(x, v)$ at the end. In contrast, a logically atomic Hoare triple $\langle x.\, P(x)\rangle\, e\, \langle v.\, Q(x, v)\rangle$ asserts that at the atomic commit point instruction of $e$, the *current* state is *some* $x$, and the instruction is given $P(x)$ and transforms it to $Q(x, v)$ at that point. That is, $e$ can only temporarily access the precondition at each instruction with the current state $x$ of that point, which may not be under full control of the current thread executing $e$.

By moving out $\mathsf{HPSlot}(tid, \_)$ to the left-hand side of a wand in PROTECT-FIXED, `protect()` is given the ownership of $\mathsf{HPSlot}(tid, \_)$ at the beginning and thus has full control of the slot's value during its execution. This reflects the property that *tid*'s slot is updated only by *tid*. On the other hand, it accesses $src \mapsto \ell * \mathsf{Managed}(\ell, v)$ only temporarily at each instruction, where $\ell$ and $v$ are current values at that point. This reflects that *src* and $\ell$ are shared locations in the data structure that can be modified by other threads.

***Proving logically atomic Hoare triples.*** To discuss how to prove a logically atomic Hoare triple, we first need to understand how it models the atomic transformation of the precondition to the postcondition. Logically atomic triples are encoded as ordinary Hoare triples with some special resources in precondition and postcondition:

$$\langle x.\, P(x)\rangle\; e\; \langle v.\, Q(x,v)\rangle := \forall R.\, \{\mathsf{AU}(P,Q,R)\}\; e\; \{v.\, R(v)\}$$

The precondition $\mathsf{AU}(P,Q,R)$, called *atomic update*, represents the permission to access the atomic precondition $P(x)$ at each atomic instruction and the obligation to transform it into the atomic postcondition $Q(x,v)$ at one of such atomic instructions. The transformation should be performed exactly once, Concretely, atomic updates can be used with the following rules:

$$\text{(AU-COMMIT)}\quad \frac{\{\exists x.\, P(x)\}\; e'\; \{v.\, Q(x,v)\} \qquad \text{physically atomic } e'}{\{\mathsf{AU}(P,Q,R)\}\; e'\; \{v.\, R(v)\}}$$

$$\text{(AU-PEEK)}\quad \frac{\{\exists x.\, P(x)\}\; e'\; \{v.\, P(x)\} \qquad \text{physically atomic } e'}{\{\mathsf{AU}(P,Q,R)\}\; e'\; \{v.\, \mathsf{AU}(P,Q,R)\}}$$

AU-COMMIT says that the atomic update can be "committed" at a physically atomic instruction. To commit the update, the instruction is given $P(x)$ at some $x$ and should transform it into $Q(x,v)$. Then, the "receipt" $R(v)$ is returned in exchange. The receipt $R(v)$ encodes the idea that a logically atomic program must commit the corresponding atomic update during its execution. By universally quantifying $R$ in the definition of the logically atomic triple, the only way to return $R(v)$ as the precondition is to commit the atomic update. For example, in the proof of stack `pop()`, this rule is used at the successful CAS to commit a successful pop operation.

The atomic update comes with another rule AU-PEEK to "peek" at the atomic precondition without actually committing the update. The atomic instruction should return the precondition without change, and then it will be returned the atomic update so that it can be used later. For example, in `pop()`, this rule is used when reading the head pointer.

***Proof of* PROTECT-FIXED.** Finally, PROTECT-FIXED can be rewritten as follows:

$$\begin{aligned}
P_{\mathtt{protect}}(src,\ell,v) &:= src \mapsto \ell * \mathsf{Managed}(\ell,v)\\
Q_{\mathtt{protect}}(src,tid,\ell,v) &:= src \mapsto \ell * \mathsf{Managed}(\ell,v) * \mathsf{Protected}(tid,v)\\
\mathsf{AU}_{\mathtt{protect}}(R) &:= \mathsf{AU}(\lambda\ell,v.\, P_{\mathtt{protect}}(src,\ell,v),\ \lambda\ell,v.\, Q_{\mathtt{protect}}(src,tid,\ell,v),\ R)
\end{aligned}$$

$$\forall src, tid, \ell_0, R.\, \{\mathsf{AU}_{\mathtt{protect}}(R) * \mathsf{HPSlot}(tid,\ell_0)\}\; \mathtt{protect}(tid,src)\; \{\ell.\, R(\ell)\}\ .$$

Fig. 13 shows the excerpt of the proof.

- V0: We start with the atomic update and the ownership of the slot.
- V1.1-2.2: We use AU-PEEK to access the atomic precondition, and read from $src \mapsto \ell_1$.
- V3.1: We generalize the precondition over the slot's value as the loop invariant.
- V3.1-3.2: We write to the thread's protected pointer slot.
- V4.1-5.3: We use either AU-COMMIT or AU-PEEK based on the value $\ell_2$.[13]
  - If $\ell_1 = \ell_2$, the validation is successful. We use AU-COMMIT to read from $src$, use HPSLOT-VALIDATE to transform HPSlot to Protected, and commit.
  - If $\ell_1 \neq \ell_2$, the validation fails. We use AU-PEEK to read from $src$. We take back the atomic update and retry (omitted).

[13]In Iris, the user actually can decide whether to commit or peek after accessing the atomic precondition.

```
V0    $\{\mathsf{AU}_{\mathtt{protect}}(R) * \mathsf{HPSlot}(tid, \ell_0)\}$
C1    fun protect($tid$, $src$):
V1.1    $\{\mathsf{AU}_{\mathtt{protect}}(R) * \mathsf{HPSlot}(tid, \ell_0)\}$
V1.2    $\{\exists \ell_1, v_1.\, src \mapsto \ell_1 * \mathsf{Managed}(\ell_1, v_1) * \mathsf{HPSlot}(tid, \ell_0)\}$
C2      $\ell_1$ := *$src$
V2.1    $\{src \mapsto \ell * \mathsf{Managed}(\ell, v) * \mathsf{HPSlot}(tid, \ell_0)\}$
V2.2    $\{\mathsf{AU}_{\mathtt{protect}}(R) * \mathsf{HPSlot}(tid, \ell_0)\}$
C3      loop:
V3.1      $\{\exists \ell'.\, \mathsf{AU}_{\mathtt{protect}}(R) * \mathsf{HPSlot}(tid, \ell')\}$
C4        protected[$tid$] := $\ell_1$
V4.1      $\{\mathsf{AU}_{\mathtt{protect}}(R) * \mathsf{HPSlot}(tid, \ell_1)\}$
V4.2      $\{\exists \ell_2, v_2.\, src \mapsto \ell_2 * \mathsf{Managed}(\ell_2, v_2) * \mathsf{HPSlot}(tid, \ell_1)\}$
C5        $\ell_2$ := *$src$
V5.1      $\{src \mapsto \ell_2 * \mathsf{Managed}(\ell_2, v_2) * \mathsf{HPSlot}(tid, \ell_1)\}$
V5.2      $\left\{ src \mapsto \ell_2 * \mathsf{Managed}(\ell_2, v_2) * \bigvee \left\{ \begin{array}{l} \ell_1 = \ell_2 * \mathsf{Protected}(tid, \ell_1, v_2) \\ \ell_1 \neq \ell_2 * \mathsf{HPSlot}(tid, \ell_1) \end{array} \right. \right\}$
V5.3      $\left\{ \bigvee \left\{ \begin{array}{l} \ell_1 = \ell_2 * R(\ell_2) \\ \ell_1 \neq \ell_2 * \mathsf{AU}_{\mathtt{protect}}(R) \end{array} \right. \right\}$
C6        ...
```

Fig. 13. Proof of Protect-Fixed

## B DETAILS OF EPOCH-BASED RCU

Among many variants of RCU [Desnoyers et al. 2012; McKenney and Slingwine 1998; McKenney et al. 2023], we focus on an epoch-based lock-free version [Fraser 2004; Harris 2001; Hart et al. 2007], and especially, the variant due to Parkinson et al. [2017]. We believe the verification techniques we propose apply also to other variants of RCU because they share the same high-level idea.

### B.1 Implementation of Epoch-Based RCU

We review the implementation of epoch-based RCU due to Parkinson et al. [2017]. To keep the algorithm simple, we assume that `retire()` is only called inside a critical section.

Fig. 14 illustrates the synchronization protocol of epoch-based RCU to ensure safe dereference of a shared pointer, $\ell$, inside a critical section delimited by a pair of `rcu_lock()` and `rcu_unlock()` invocations.[14] (Please ignore the boxes for now.) The critical sections synchronize with each other using *epoch counters*. In the figure, black arrows represent four critical sections for threads $T_0$ to $T_3$ that are *locked at epochs* 10, 11, 12, and 13, respectively.

The epoch-based RCU has the following *epoch-consensus protocol*: the difference between the locked epochs of concurrently active critical sections is at most one. Specifically, as represented by

[14]For concise presentation, assume that an address coincides with the block ID. Our formalized verification lifts this assumption by tracking the current block ID of each address, as briefly noted in §5.

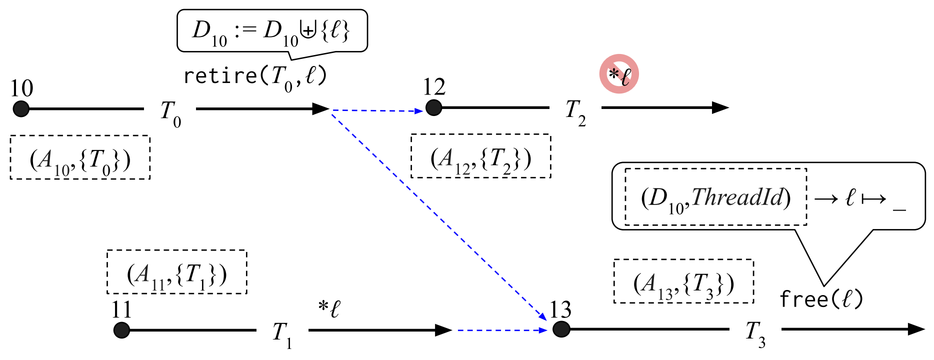


Fig. 14. The epoch consensus example annotated with tokens.

the blue dashed arrows, the end of a critical section locked at an epoch, say 10, happens before the beginning of another critical section locked at 12 or later.

Under this protocol, a retired memory block is can be reclaimed after three epochs have passed. For instance, since thread $T_0$ retires $\ell$ at the epoch 10, it can be reclaimed by thread $T_3$ at the epoch 13. This reclamation rule is safe thanks to the following property: a retired pointer becomes inaccessible to all threads after two epochs have passed. For instance, $T_2$ locked at 12 cannot access $\ell$ retired by $T_0$ locked at 10. Formally:

Lemma 1. *Let $D_e$ be the set of retired pointers at epoch $e$. If a pointer is in $\cup_{i \le e} D_i$, then it is inaccessible at epoch $e + 2$.*

Proof. Let $\ell \in D_i$ be a pointer retired at the epoch $i$. Then $\ell$ is detached before the end of a critical section locked at $i$, which happens before the beginning of a critical section locked at $e + 2$, as $i + 2 \le e + 2$. Thus, a critical section locked at $e + 2$ cannot access the detached pointer $\ell$. □

As a consequence, retired pointers are indeed safe to reclaim after three epochs have passed, *e.g.*, $T_3$ locked at 13 can reclaim $\ell$. If $\ell$ is retired at $e$, then all accesses to $\ell$ are in the critical sections locked at $i \le e + 1$, which happens before the beginning of the critical sections locked at $e + 3$ or later. A natural corollary of Lemma 1 is that the complement of retired sets until $e - 2$, $Loc \setminus \cup_{i \le e-2} D_i$, is precisely the set of pointers a thread locked at $e$ may access. We denote this set by $A_e$.

Fig. 15 presents an implementation of RCU based on the above epoch consensus rule. The global variable `global_epoch` contains the *monotonic global epoch* and `locked_epochs` is the list of the epochs at which each thread is locked. For each $e$, we maintain invariants that **(1)** the increment of the global epoch to $e$ happens before the beginning of a critical section locked at $e$; and **(2)** the end of a critical section locked at $e$ happens before the increment of global epoch to $e + 2$. As a consequence, the end of a critical section locked at $e$ happens before the increment of the global epoch to $e + 2$, which in turn happens before the beginning of a critical section locked at $e + 2$, satisfying the epoch consensus.

The implementation resembles hazard pointers in the sense that `rcu_lock(tid)` proposes an epoch it wants to protect, and `try_advance()` checks each thread's epoch before incrementing the global epoch. Specifically, `rcu_lock(tid)` copies the global epoch to its locked epoch slot (lines C12-14); validates that global epoch has not changed (lines C15-17); and if changed, retries with the new epoch (line C13). The validation is necessary for maintaining the epoch consensus: if the global

```
C11 fun rcu_lock(tid):
C12   e := global_epoch
C13   loop:
C14     locked_epochs[tid] := e
C15     e' := global_epoch
C16     if e' == e:
C17       return
C18     e := e'

C21 fun rcu_unlock(tid):
C22   locked_epochs[tid] := -1

C31 fun retire(tid, p):
C32   retired.push((p, locked_epochs[tid]))
C33   if /* some condition */:
C34     do_reclamation()
```

```
C41 fun do_reclamation():
C42   ge := try_advance()
C43   for (r, e) in retired.pop_all():
C44     if ge < e + 3:
C45       retired.push((r, e))
C46     else:
C47       free(r)

C51 fun try_advance():
C52   ge := global_epoch
C53   for e in locked_epochs:
C54     if e ≥ 0 && e ≠ ge:
C55       return ge
C56   if CAS(&global_epoch, ge, ge + 1):
C57     return ge + 1
C58   else:
C59     return ge
```

Fig. 15. Implementation of epoch-based RCU.

epoch has increased between C12-14, the incrementing thread has not seen `tid`'s intention to begin a critical section, breaking the second invariant. To end the protection, `rcu_unlock(tid)` simply assigns the dummy value $-1$ to `tid`'s epoch slot (line C22). On the other hand, `try_advance()` tries incrementing the global epoch by one (line C56-59) only if every active critical section is locked at the same epoch as the global epoch (line C52-55).

The `retire(tid, p)` and `do_reclamation()` functions are similar to those of hazard pointers, but with the following differences: **(1)** a retired pointer is annotated with the epoch at which the enclosing critical section is locked (line C32); and **(2)** a retired pointer is reclaimed only if three epochs have passed since the retirement (line C44).

## B.2 Verification of Epoch-Based RCU

We sketch a proof that the implementation of RCU (Fig. 15) satisfies the epoch consensus (Fig. 14) and our specification (Fig. 11).

***Reserving Pointer Ownership with Permission Tokens.*** Unlike hazard pointers, RCU's protection rule MANAGED-PROTECTED does not require any physical interaction with the internals of RCU. Instead, the synchronization of `rcu_lock()` at epoch $e$ alone guarantees safe access to all pointers in $A_e$. This means that ownership transfer of the entire $A_e$ must happen during RCU-LOCK, and MANAGED-PROTECTED just recalls that the guard has already obtained the ownership for the pointer, by assuring that it is not retired at $e$, hence in $A_e$.

However, at RCU-LOCK time, some pointers in $A_e$ may have not been allocated yet. For example, a thread in a critical section may access a memory block just allocated and attached to a data structure after it entered the critical section. Thus, RCU-LOCK cannot take the necessary partial points-to ownership of those non-existent pointers.

$$\mathsf{EpochHistory}(D : \mathit{List}(\wp(\mathit{Loc}))) : \mathit{iProp} \qquad \mathsf{OwnEpoch}(e : \mathbb{N},\ A_e : \wp(\mathit{Loc}),\ s : \wp(\mathit{ThreadId})) : \mathit{iProp}$$

(Epoch-History-Retire)

$$\mathsf{EpochHistory}(D) * \mathsf{OwnEpoch}(e, A_e, \{tid\}) \Rrightarrow \mathsf{EpochHistory}(D[e \mapsto D[e] \uplus \{\ell\}]) * \mathsf{OwnEpoch}(e, A_e, \{tid\})$$

(Epoch-Own-Fract)

$$\mathsf{OwnEpoch}(e, A_e, s_1) * \mathsf{OwnEpoch}(e, A_e, s_2) \dashv\vdash \mathsf{OwnEpoch}(e, A_e, s_1 \uplus s_2)$$

(Epoch-History-Advance)

$$\textbf{let } e := \mathsf{len}(D) - 1 \textbf{ in}$$

$$\mathsf{EpochHistory}(D) \Rrightarrow \mathsf{EpochHistory}(D +\!\!+ [\emptyset]) * \mathsf{OwnEpoch}(e + 1, \mathit{Loc} \setminus \textstyle\bigcup_{i \le e-1} D[i], \mathit{ThreadId})$$

$$\mathsf{RCUInv} := \boxed{\begin{array}{l}\exists D.\, \textbf{let } e := \mathsf{len}(D) - 1 \textbf{ in } \mathtt{global\_epoch} \mapsto e * \mathtt{locked\_epochs} \xmapsto{1/2} \ldots * \\ \quad \mathsf{EpochHistory}(D) * (\text{mutual exclusion protocol governing OwnEpoch and tokens}) * \ldots\end{array}}$$

$$\mathsf{Guard}(tid, G) := \exists e.\, \mathtt{locked\_epochs}[tid] \xmapsto{1/2} e * \mathsf{OwnEpoch}(e, A_e, tid) * \boxed{(A_e, \{tid\})} * G \subseteq A_e * \ldots$$

Fig. 16. Definition of the predicates and invariant of epoch-based RCU.

We follow Tassarotti et al. [2015] to resolve this issue by introducing a logical proxy of pointer to reserve the pointer ownership. Specifically, they use *permission tokens* of the form $\boxed{(\{\ell\}, \{tid\})} \in \wp(\mathit{Loc})_{\uplus} \times \wp(\mathit{ThreadId})_{\uplus}$ which are ghost resources allocated at the beginning of the proof, and set up invariants that each token can be exchanged with the actual pointer ownership $\ell \xmapsto{\{tid\}} \_$.

***Distributing and Collecting Tokens.*** Now we verify RCU's proof rules (Fig. 11) by reasoning about the permission token's transfer.

We first reason about the contention between RCU-Lock and `try_advance()` on the ownership of tokens. This interaction can be understood as a mutual exclusion protocol, just like the contention between `protect()` and `do_reclamation()` in hazard pointers (§4.2). Specifically, the `locked_epochs` and `global_epoch` variables play the role of $p$ and $d$ flags in Fig. 8, respectively. On the one hand, when `rcu_lock(`$tid$`)` validates epoch $e$, it takes tokens $\boxed{(A_e, \{tid\})}$ that are returned by `rcu_unlock(`$tid$`)`, as illustrated in Fig. 14. On the other hand, when `try_advance()` increases the global epoch from $e - 1$ to $e$, the value of each `locked_epochs[`$tid$`]` must've been -1 or $e - 1$. At that point, $tid$ must have returned tokens for $A_{e-2} \setminus A_{e-1} = D_{e-3}$, which means that `try_advance()` may take those tokens.

To verify Managed-Protected, we prove that the guard already has the token of the pointer $\ell$ to protect. To this end, we maintain an invariant that Managed$(\ell, _)$ implies $\ell \notin \bigcup_{i \in \mathbb{N}} D_i$. We also note that $D_i$ monotonically grows over time for each $i$, so if a pointer is currently not in $\bigcup_{i \in \mathbb{N}} D_i$, then it must have not been there in the past either. Thus, if one can provide Managed$(\ell, _)$ to Managed-Protected, $A_e$ at RCU-Lock contains $\ell$, and thus the Guard$(tid, _)$ has the token for $\ell$.

***Tracking Retired Pointers with Epoch History Ghost State.*** In the above verification, we did not formalize how the logical variable $(D_i)_{i \in \mathbb{N}}$ is modified by multiple threads. To fill in the gap, we maintain a protocol on $(D_i)_i$ with the *epoch history* ghost state, illustrated in Fig. 16, that consists of two resource types:

- EpochHistory$(D)$, the authoritative resource owned by the RCU's invariant RCUInv to record the up-to-date list of set of pointers retired at each epoch; and

- $\mathsf{OwnEpoch}(e, A_e, \{tid\})$, a resource owned by $\mathsf{Guard}(tid, \_)$ that represents the fractional permission for $tid$ (Epoch-Own-Fract) to retire pointers at epoch $e$ (Epoch-History-Retire) and the knowledge that $A_e$ is accessible in epoch $e$.

In the invariant RCUInv of epoch-based RCU, we also relate the epoch history ghost state and the physical value of `global_epoch` by equating the last index of $D$ in $\mathsf{EpochHistory}(D)$ and `global_epoch`. The mutual exclusion protocol between RCU-Lock and `try_advance()` described above also governs transfer of $\mathsf{OwnEpoch}(e, A_e, \{tid\})$. When `try_advance()` increments epoch, it uses Epoch-History-Advance to allocate the ownership for the new epoch and compute the pointers accessible in a critical section of the new epoch.

## C APPLICATION TO MORE SMR SCHEMES

We sketch a specification for DEBRA+ [Brown 2015], PEBR [Kang and Jung 2020], and NBR [Singh et al. 2021]. These schemes are a hybrid of RCU and hazard pointer designed to cope with the original schemes' weaknesses:

- RCU's lack of robustness: if any of the threads do not exit their critical section, the reclamation of retired nodes cannot progress;
- hazard pointers' incompatibility with many data structures: it does not apply to data structures that use optimistic traversal, such as Harris's list.

They use a mechanism called *neutralization* or *ejection* to deal with RCU's robustness issue: if there is a non-cooperative thread, the scheme forcefully ends the thread's critical section so that reclamation can proceed. These schemes apply to optimistic traversal because the traversal is aborted only when it is absolutely necessary. When a thread detects that it is neutralized, it should stop traversing and start a recovery procedure. To help the recovery procedure, those schemes use hazard pointers to keep protecting (outside the critical section) the memory blocks relevant for recovery. Of course, this requires the thread to manually announce the protection of such pointers.

The protection of these schemes can be modeled with a common function `try_protect(tid, p)`. Inside a critical section, it tries announcing protection of `p`, and returns a boolean value that indicates whether it was successful. **`true`** means that the protection is established and validated. **`false`** means that the protection failed and the critical section was neutralized. In this case, the thread has to run the recovery procedure only using the pointers that have been successfully protected. PEBR provides a function similar to `try_protect()`. In contrast, DEBRA+ and NBR use POSIX signals for detecting ejection and non-local goto for entering the recovery procedure. Despite the superficial difference, `try_protect()` still captures the protection of DEBRA+ and NBR, because signal handlers can be modeled by atomically checking the signal on each step of execution.

`try_protect()` can be specified in a style that resembles our specifications for both hazard pointer and RCU.

(Try-Protect)

$$\frac{i \notin R}{\mathsf{BlockInfo}(i, \ell, P) \vdash \{\mathsf{Guard}(tid, G, R)\}\ \mathtt{try\_protect}(tid, \ell)\ \left\{b.\ \bigvee \begin{cases} b \land \mathsf{Guard}(tid, G[\ell \mapsto (i, P)], R) \\ \neg b \land \mathsf{FrozenGuard}(tid, G) \end{cases}\right\}}$$

In the premise, we have a BlockInfo for the pointer we are trying to protect, and the knowledge that the pointer was not retired before the start of the critical section (*i.e.*, not in $R$). In the postcondition, if `try_protect()` was successful, we update the *guarded set* $G$ to include the newly protected pointer. Otherwise, the guard changes to a *frozen guard*. The following two key differences capture the essence of the neutralization mechanism. **(1)** The Guard predicate additionally holds a map

$G$ of the *currently protected pointers*, which records its block ID and block resource, similar to a Protected. **(2)** The FrozenGuard predicate represents the protected pointers of a neutralized thread: it can still access the pointers it has protected so far (represented by $G$), but cannot protect new pointers (represented by the fact that it no longer records $R$).

## REFERENCES

Yehuda Afek, Guy Korland, and Eitan Yanovsky. 2010. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6490)*. Springer, 395–410. https://doi.org/10.1007/978-3-642-17653-1_29

Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-Ups: Concurrency in the Linux Kernel. *SIGPLAN Not.* 53, 2 (March 2018), 405–418. https://doi.org/10.1145/3296957.3177156

Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2017. Forkscan: Conservative Memory Reclamation for Modern Operating Systems. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 483–498. https://doi.org/10.1145/3064176.3064214

Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2018. ThreadScan: Automatic and Scalable Memory Reclamation. *ACM Trans. Parallel Comput.* 4, 4, Article 18 (may 2018), 18 pages. https://doi.org/10.1145/3201897

Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2021. Concurrent Deferred Reference Counting with Constant-Time Overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 526–541. https://doi.org/10.1145/3453483.3454060

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission Accounting in Separation Logic. *SIGPLAN Not.* 40, 1 (jan 2005), 259–270. https://doi.org/10.1145/1047659.1040327

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–72.

Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing* (Donostia-San Sebastián, Spain) *(PODC '15)*. Association for Computing Machinery, New York, NY, USA, 261–270. https://doi.org/10.1145/2767386.2767436

David Chase and Yossi Lev. 2005. Dynamic Circular Work-Stealing Deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Las Vegas, Nevada, USA) *(SPAA '05)*. Association for Computing Machinery, New York, NY, USA, 21–28. https://doi.org/10.1145/1073970.1073974

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–231.

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt Meets Relaxed Memory. *Proc. ACM Program. Lang.* 4, POPL, Article 34 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371102

Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: Strong and Compositional Library Specifications in Relaxed Memory Separation Logic. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 792–808. https://doi.org/10.1145/3519939.3523451

John Derrick, Brijesh Dongol, Gerhard Schellhorn, Bogdan Tofan, Oleg Travkin, and Heike Wehrheim. 2014. Quiescent Consistency: Defining and Verifying Relaxed Linearizability. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8442)*. Springer, 200–214. https://doi.org/10.1007/978-3-319-06410-9_15

M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (2012), 375–382. https://doi.org/10.1109/TPDS.2011.159

Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2004. Formal Verification of a Practical Lock-Free Queue Algorithm. In *Formal Techniques for Networked and Distributed Systems – FORTE 2004*, David de Frutos-Escrig and Manuel Núñez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–114. https://doi.org/10.1007/978-3-540-30232-2_7

Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 448–475.

Keir Fraser. 2004. *Practical lock-freedom.* Ph. D. Dissertation.

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–402. https://doi.org/10.1007/978-3-642-15375-4_27

Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 249–269. https://doi.org/10.1007/978-3-642-37036-6_15

Alexey Gotsman and Hongseok Yang. 2012. Linearizability with Ownership Transfer. In *CONCUR 2012 – Concurrency Theory*, Maciej Koutny and Irek Ulidowski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–271. https://doi.org/10.1007/978-3-642-32940-1_19

Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. 2016. Local Linearizability for Concurrent Container-Type Data Structures. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada (LIPIcs, Vol. 59)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.6

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-Word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing (DISC '02)*. Springer-Verlag, Berlin, Heidelberg, 265–279.

Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of Memory Reclamation for Lockless Synchronization. *J. Parallel Distrib. Comput.* 67, 12 (dec 2007), 1270–1285. https://doi.org/10.1016/j.jpdc.2007.04.010

Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A Scalable Lock-Free Stack Algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Barcelona, Spain) *(SPAA '04)*. Association for Computing Machinery, New York, NY, USA, 206–215. https://doi.org/10.1145/1007912.1007944

Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. 2013. Quantitative relaxation of concurrent data structures. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. ACM, 317–328. https://doi.org/10.1145/2429069.2429109

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

Iris Team. 2023a. Iris examples. https://gitlab.mpi-sws.org/iris/examples

Iris Team. 2023b. The Iris project website. https://iris-project.org/

Bart Jacobs and Frank Piessens. 2011. Expressive Modular Fine-Grained Concurrency Specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 271–282. https://doi.org/10.1145/1926385.1926417

Radha Jagadeesan and James Riely. 2014. Between Linearizability and Quiescent Consistency - Quantitative Quiescent Consistency. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8573)*. Springer, 220–231. https://doi.org/10.1007/978-3-662-43951-7_19

Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. 2023. Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic (Coq development and appendix). https://doi.org/10.1145/3580418 Project webpage: https://cp.kaist.ac.kr/gc.

Ralf Jung. 2019. Logical Atomicity in Iris: the Good, the Bad, and the Ugly. Iris Workshop. https://people.mpi-sws.org/~jung/iris/talk-iris2019.pdf

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2019. The Future is Ours: Prophecy Variables in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 45 (dec 2019), 32 pages. https://doi.org/10.1145/3371113

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. *SIGPLAN Not.* 52, 1 (jan 2017), 175–189. https://doi.org/10.1145/3093333.3009850

Jeehoon Kang and Jaehwang Jung. 2020. A Marriage of Pointer- and Epoch-Based Reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 314–328. https://doi.org/10.1145/3385412.3385978

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/3009837.3009855

Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2017. Go with the Flow: Compositional Abstractions for Concurrent Data Structures. *Proc. ACM Program. Lang.* 2, POPL, Article 37 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158125

Ismail Kuru and Colin S. Gordon. 2019. Safe Deferred Memory Reclamation with Types. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 88–116. https://doi.org/10.1007/978-3-030-17184-1_4

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Jean-Marie Madiot and François Pottier. 2022. A Separation Logic for Heap Space under Garbage Collection. *Proc. ACM Program. Lang.* 6, POPL, Article 11 (jan 2022), 28 pages. https://doi.org/10.1145/3498672

Paul E. McKenney, Maged Michael, Jens Maurer, Peter Sewell, Martin Uecker, Hans Boehm, Hubert Tong, Niall Douglas, Thomas Rodgers, Will Deacon, Michael Wong, David Goldblatt, Kostya Serebryany, and Anthony Williams. 2021. P2414R1: Pointer lifetime-end zap proposed solutions. https://wg21.link/p2414r1.

P. E. McKenney and J. D. Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *PDCS '98*.

Paul E. McKenney, Michael Wong, Maged M. Michael, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, Daisy Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Erik Rigtorp, Tomasz Kamiński, and Jens Maurer. 2023. P2545R4: Read-Copy Update (RCU). https://wg21.link/p2545r4.

Meta. 2023. Folly: Facebook Open-source Library. https://github.com/facebook/folly

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: A Concurrent Separation Logic for Multicore OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 96 (aug 2020), 29 pages. https://doi.org/10.1145/3408978

Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022. A Concurrent Program Logic with a Future and History. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 174 (oct 2022), 30 pages. https://doi.org/10.1145/3563337

Roland Meyer and Sebastian Wolff. 2019a. Decoupling Lock-Free Data Structures from Memory Reclamation for Static Analysis. *Proc. ACM Program. Lang.* 3, POPL, Article 58 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290371

Roland Meyer and Sebastian Wolff. 2019b. Pointer Life Cycle Types for Lock-Free Data Structures with Memory Reclamation. *Proc. ACM Program. Lang.* 4, POPL, Article 68 (dec 2019), 36 pages. https://doi.org/10.1145/3371136

Maged Michael, Maged M. Michael, Michael Wong, Paul McKenney, Andrew Hunter, Daisy S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, and Mathias Stearn. 2023. P2530R3: Hazard Pointers for C++26. https://wg21.link/p2530r3.

Maged M. Michael. 2002. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Winnipeg, Manitoba, Canada) *(SPAA '02)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/564870.564881

Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (June 2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC 1996*.

Ike Mulder and Robbert Krebbers. 2023. Proof Automation for Linearizability in Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 91 (apr 2023), 30 pages. https://doi.org/10.1145/3586043

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 809–824. https://doi.org/10.1145/3519939.3523432

Ruslan Nikolaev and Binoy Ravindran. 2020. *Universal Wait-Free Memory Reclamation*. Association for Computing Machinery, New York, NY, USA, 130–143. https://doi.org/10.1145/3332466.3374540

Ruslan Nikolaev and Binoy Ravindran. 2021. Brief Announcement: Crystalline: Fast and Memory Efficient Wait-Free Reclamation. In *35th International Symposium on Distributed Computing (DISC 2021) (Leibniz International Proceedings in*

*Informatics (LIPIcs), Vol. 209)*, Seth Gilbert (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 60:1–60:4. https://doi.org/10.4230/LIPIcs.DISC.2021.60

Matthew Parkinson, Richard Bornat, and Peter O'Hearn. 2007. Modular Verification of a Non-Blocking Stack. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 297–302. https://doi.org/10.1145/1190216.1190261

Matthew Parkinson, Kapil Vaswani, Dimitrios Vytiniotis, Manuel Costa, Pantazis Deligiannis, Aaron Blankstein, Dylan McDermott, and Jonathan Balkind. 2017. *Project Snowflake: Non-blocking safe manual memory management in .NET*. Technical Report MSR-TR-2017-32. Microsoft. https://www.microsoft.com/en-us/research/publication/project-snowflake-non-blocking-safe-manual-memory-management-net/

Gali Sheffi, Maurice Herlihy, and Erez Petrank. 2021. VBR: Version Based Reclamation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 443–445. https://doi.org/10.1145/3409964.3461817

Ajay Singh, Trevor Brown, and Ali Mashtizadeh. 2021. NBR: Neutralization Based Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 175–190. https://doi.org/10.1145/3437801.3441625

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*. Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying Read-Copy-Update in a Logic for Weak Memory. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 110–120. https://doi.org/10.1145/2737924.2737992

Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif. 2011. Formal Verification of a Lock-Free Stack with Hazard Pointers. In *Theoretical Aspects of Computing – ICTAC 2011*, Antonio Cerone and Pekka Pihlajasaari (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 239–255. https://doi.org/10.1007/978-3-642-23283-1_16

R. K. Treiber. 1986. Systems programming: coping with parallelism.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 691–707. https://doi.org/10.1145/2660193.2660243

Viktor Vafeiadis. 2010a. Automatically Proving Linearizability. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

Viktor Vafeiadis. 2010b. RGSep Action Inference. In *Verification, Model Checking, and Abstract Interpretation*, Gilles Barthe and Manuel Hermenegildo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 345–361. https://doi.org/10.1007/978-3-642-11319-2_25

Viktor Vafeiadis. 2011. Concurrent Separation Logic and Operational Semantics. *Electronic Notes in Theoretical Computer Science* 276 (2011), 335–351. https://doi.org/10.1016/j.entcs.2011.09.029 Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

Viktor Vafeiadis. 2017. Program Verification Under Weak Memory Consistency Using Separation Logic. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 30–46.

Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-Based Memory Reclamation. *SIGPLAN Not.* 53, 1 (feb 2018), 1–13. https://doi.org/10.1145/3200691.3178488

Sebastian Wolff. 2021. *Verifying Non-blocking Data Structures with Manual Memory Management.* Ph. D. Dissertation. https://doi.org/10.24355/dbbs.084-202108191157-0

Received 2023-04-14; accepted 2023-08-27