Verifying Read-Copy-Update in a Logic for Weak Memory

Joseph Tassarotti
Carnegie Mellon University
tassaro@andrew.cmu.edu

Derek Dreyer
MPI-SWS, Germany
dreyer@mpi-sws.org

Viktor Vafeiadis
MPI-SWS, Germany
viktor@mpi-sws.org

Abstract

Read-Copy-Update (RCU) is a technique for letting multiple readers safely access a data structure while a writer concurrently modifies it. It is used heavily in the Linux kernel in situations where fast reads are important and writes are infrequent. Optimized implementations rely only on the weaker memory orderings provided by modern hardware, avoiding the need for expensive synchronization instructions (such as memory barriers) as much as possible.

Using GPS, a recently developed program logic for the C11 weak memory model, we verify an implementation of RCU for a singly-linked list assuming "release-acquire" semantics. This is, to our knowledge, the first formal proof of an RCU implementation in a weak-memory setting.

1. Introduction

Traditionally, most work on concurrent program verification has assumed a sequentially consistent (SC) model of memory, in which updates to memory are globally visible to all threads as soon as they occur [Adve and Gharachorloo 1996]. For performance reasons, however, modern architectures offer weaker guarantees about the ordering of concurrent memory operations [Sewell et al. 2010; Mador-Haim et al. 2012]. Although it is possible to simulate SC semantics on such hardware by inserting explicit synchronization instructions (e.g., barriers/fences), the cost of doing so—particularly for high-performance concurrent code—can be prohibitive.

Fortunately, for many concurrent algorithms, full SC behavior is unnecessary, and more limited forms of synchronization suffice. One widely-used example is Read-Copy-Update (RCU) [McKenney and Slingwine 1998; McKenney 2004]. RCU is a technique, deployed heavily in the Linux kernel, that lets a single writer manipulate a data structure while multiple readers are concurrently accessing it. Instead of directly modifying a piece of the structure, the writer first copies that piece, modifies the copy, and then makes the new copy accessible and the old one inaccessible. Once no readers are capable of accessing the old copy, the writer may safely deallocate it. However, until that time, some readers may see the old copy while others see the new copy, and there is no guarantee when readers will begin to see the new copy.

As this description suggests, RCU employs some synchronization (e.g., to ensure memory safety), but not full SC semantics, and its reliance on weaker memory assumptions is essential to its efficiency. However, the only existing formal proof of correctness for an RCU-based data structure [Gotsman et al. 2013] assumes an SC memory model.

In this paper, we give the first formal proof of correctness for an implementation of RCU under a weak memory model. Specifically, we verify a user-space RCU implementation of linked lists (based on that of Desnoyers et al. [2012]), programmed using release-acquire atomics, one of the main weak-memory access modes supported by the C/C++11 language standard [ISO/IEC 9899:2011].

Why focus on release-acquire? There are several reasons. First and foremost, the semantics of C11’s release-acquire mode has been fully formalized [Batty et al. 2011], rendering our RCU implementation amenable to formal verification. Second, release-acquire semantics, while significantly weaker than SC, nevertheless provides sufficiently strong synchronization to guarantee safety of our RCU implementation. Third, release-acquire semantics is “reasonable”, in the sense that one can reason about it using a more restricted version of the kinds of reasoning principles that hold under SC semantics. This claim was substantiated formally by recent work of Turon et al. [2014] on a logic called GPS, which supports Hoare-style verification of programs that use C11’s release-acquire atomics. Here, we leverage (a mild extension of) GPS in our verification, while simultaneously demonstrating a much more significant case study for the use of GPS than any that Turon et al. previously considered.

Above and beyond our formal verification of RCU in GPS (described in full formal detail in our technical appendix), an important contribution of this work is the proof idea itself, and it is the elucidation of this proof idea that is our main goal in this paper. Previously, the correctness of RCU has been argued using the concept of a grace period during which reader threads may finish accessing an old node before it is deallocated. Indeed, Gotsman et al.’s proof in the SC setting depends on an extension of separation logic with a temporal “since” operator to formalize grace periods. In contrast, our proof avoids any such extension; we rely instead on GPS’s

1 The expert reader may wonder why we do not consider C11’s release-consume semantics instead. Please see the discussion of this point in §7.
notion of per-location protocols, which describe how the state of a shared memory location may evolve over time. Using per-location protocols, Turon et al. showed how to formalize the folklore intuition that release-writes and acquire-reads support a form of message passing between threads. We in turn show how such message passing provides a novel and self-sufficient explanation of how RCU works—e.g., of how the writer tells the readers which nodes have been deallocated, or how the readers tell the writer that they will no longer access them. No additional mechanism is required.

In the rest of the paper, we present the semantics of release-acquire (§2), our implementation of RCU and why it intuitively works (§3), our specification of RCU and why it is useful (§4), a review of GPS with some minor extensions (§5), a high-level description of our verification of RCU in GPS (§6), and discussion of related work (§7).

2. Release-Acquire Semantics

Our implementation of RCU uses a simple imperative while-language extended with C11’s release-acquire memory operations. In this section, we try to give some intuition for the semantics of these operations. Thorough, formal presentations of the C11 memory model can be found in Batty et al. [2011] and Vafeiadis et al. [2015].

The C11 standard divides memory accesses into two types—atomic and non-atomic—and we will say that a memory location is atomic (resp. non-atomic) if all accesses to it are of that type. Non-atomic accesses are the default variety, appropriate for most use cases: they are fast and do not require the compiler to emit special synchronization instructions, but the standard says that the behavior of a program is undefined if it involves racy non-atomic accesses (i.e., two unordered non-atomic accesses to a single location, at least one of which is a write). Thus, the onus is on the programmer to employ enough synchronization to ensure that there are no non-atomic data races.

One way of implementing such synchronization is using atomic locations, which are intended for racy access. The C11 standard lets programmers annotate reads and writes for atomic locations with different consistency level options, ranging from sequentially consistent (SC) to fully relaxed, depending on how cheap they want the accesses to be and how much instruction reordering they can tolerate.

We focus here on only two of these options: release writes and acquire reads. When a thread performs a release write, and another thread observes that write via an acquire read, the operations that precede the write are guaranteed to “happen-before” the read. This kind of “transitive visibility”, as it is often called, allows programmers to use release-acquire to implement a message-passing idiom.

For example, consider the following contrived yet illustrative code (which we will show how to verify in §5):

```c
[x]na := 37; [y]na := 25;
[y]rel := 1; if ([y]acq == 1) {
  [y]na := [x]na;
}
while ([rf]acq != 1) { /* spin */
  [rf]rel := 1;
免费(x);
/* postcond: y ↔ 37 ∨ y ↔ 25 */
```

Here, we have two non-atomic memory locations, x and y, and two atomic “flags”, lf and rf. We write [x]na to indicate a non-atomic operation on location x, and [y]rel and [y]acq for release and acquire operations on lf. The code on the sides of the vertical bars represents two different threads, and initially we assume that all locations contain 0.

The effect of this code is to pass control of the non-atomic location x from the left thread to the right thread and back. The left thread initializes location x with value 37, and then sets its flag lf to 1 with a release write, to signal that the right thread can now access x. The right thread checks if lf is set to 1, and if so, it reads the value of x (non-atomically) and sets y to that value; otherwise, it sets y to 25. Because the non-atomic write of 37 to x preceded the release write to lf, and the matching acquire read of lf as 1 (if it occurred) preceded the non-atomic read of x, we will be able to establish: (1) the write to x happened before the read of x, so there is no data race on x, and (2) as a postcondition, y may either point to 25 or 37, but not 0. After the second thread finishes accessing x, it does a release write of 1 to rf to signal to the left thread that it is done. The first thread spins until it observes this write. At this point, it knows it can safely deallocate x because the second thread’s read of x must have happened already.

In the previous example, the two threads synchronized their operations by using release writes to the atomic locations lf and rf to send messages back and forth, and acquire reads to receive them. In contrast, here is an example based on the classic “Dekker’s algorithm” for mutual exclusion [Dijkstra 1965], which is safe under SC but not under release-acquire:

```c
[x]rel := 1; if ([y]acq == 0) then {
  [z]na := 1
}
[y]rel := 1; if ([x]acq == 0) then {
  [z]na := 2
}
```

Here, the left thread does a release write to set x to 1, while the right thread does the same for y. Each thread then does an acquire read of the other’s variable to see if it has been updated. If not, each thread tries to modify the non-atomic z.

Under an SC semantics, one of the thread’s writes must happen before the other: that thread “wins” and may write to z. For instance, if the left thread reads y and sees 0, it concludes that the right thread has not written to y yet, so the left thread knows it has won the race. However, under release-acquire, the writes to x and y are unordered, and it is possible for the left thread to read 0 from y and for the right thread to read 0 from x in the same execution. If this happens, they will both try to write to z, resulting in a data race.

Informally, if we think in terms of message passing, this example is unsound because it tries to conclude something from the negative fact that a message has not yet arrived.
We now describe how RCU can be implemented for a singly
linked list using the release-acquire memory operations. In
explaining the algorithm, we focus on how the orderings
imposed by pairs of release-acquire operations ensure that
there are no data races. In each case, we can informally
describe these operations in terms of how they send messages
between the threads. In §6 our proof will make this message-
passing explanation precise.

A simplified part of our verified implementation is pre-
sented in Figure 1. Nodes in the list are records with two
fields. The data field contains the contents of the node, and
link is a pointer to the next node in the list.

**Initialization** A new RCU instance is created by calling
rcuNew. This returns a pointer q to the metadata for the
RCU instance, which consists of a counter for the writer
(q + wcounter), an array of counters for the readers (q +
rcounters), which we describe below, and a field containing
a pointer to the head of the list (q + 1ink). The counters all
start at 0, and the q + 1ink field is initially a null pointer.

**Reading** The readers access the nodes in the list in a loop
that maintains a current pointer into the list. They start the
traversal by calling rcuReadStart to get the first pointer
to the head of the list. This does an acquire read on q + 1ink
and returns the result.

Then, within a loop, they check that their current pointer
is not null, and if so, access the value stored at the node
and the next node by calling rcuReadNext with their current
pointer as p. The function simply reads off the data and link
fields of p and returns the results. While the data field is
read non-atomically, the link field is accessed by an acquire
read—this ensures correct synchronization with the writer.

**Updating the list** We now explain how the writer modifies
the list by walking through an example shown in Figure 2.
It depicts the linked list, and the counters in the RCU meta-
data. The writer thread and one of the reader threads are
represented by diamonds labeled “W” and “R”. To represent
messages being passed from one thread to the other through
a location, we draw a dashed arrow from the sender to the
location and another from the location to the reader.

Suppose the writer wants to change the value in the second
node of the linked list shown in Figure 2a. To do so, it calls
rcuNodeUpdate(q, x, p, v) where q is a pointer to the RCU
metadata, x is a pointer to the node that it wants to modify, p
is a pointer to the previous node, and v is the new value for
the node. This function allocates a new node, copies the old
node’s link field, and sets the updated value as in Figure 2b.
Next, it updates the previous node’s link field with a release
write so that it points to the new node (line 16), rendering the
old node unreachable as shown in 2c. At this point, readers
may begin to see the new node when they do an acquire read
on the previous link pointer (line 8). The pairing between
this release write and the readers’ acquire reads is the first
important point of synchronization in RCU:

**Release-Acquire Pair 1 (link field)**: The order-
ing imposed by the release writes and acquire reads
on the link fields ensures that the initialization of
the node precedes the readers’ accesses. In other
words, the writer passes a message to the readers
saying that the next nodes are safe to access.

**Dealocating the old node** After completing the release
write in Figure 2c, the writer calls rcuSynchronize to wait
until no readers can access the old copy any longer, so that it
can deallocate the removed node. There are a number of ways to implement rcuSynchronize without sacrificing reader performance. The code in Figure 1 uses Quiescent State Based Reclamation (QSBR) [Desnoyers et al. 2012].

In QSBR, the writer begins the synchronization operation by incrementing its counter (line 22 and Figure 2d). It then repeatedly reads each reader counter in turn until they all match the writer counter’s new value. When readers are not accessing the list (that is, they are quiescent), they periodically call rcuQuiescentState which examines the writer’s counter and copies its value into the reader’s counter (Figure 2e). Once the writer sees that every reader’s counter matches its own, it knows they have all entered a quiescent state since the old node became unreachable. This means that if the readers access the list in the future, they will not access the old node, so the node can be safely deallocated (Figure 2f).

These counter fields must be atomic, because the readers will try to concurrently read the writer’s counter as the writer is incrementing it, and vice-versa. Both counters are involved in a release-acquire synchronization:

Release-Acquire Pair 2 (wcounter field): The synchronization between the write on line 22 and the read on line 10 guarantees that once the reader sees the updated counter, it will see the update that made the old node unreachable (line 16). When the writer increments its counter, it publishes the fact that a node has been made unreachable together with a request for permission to deallocate the node.

Release-Acquire Pair 3 (rcounters + tid field): When the readers update their counters to match the writer’s on line 11, they acknowledge the writer’s request by giving up their own permission to access the unreachable node. The release-acquire ordering ensures that any accesses the reader was doing before calling rcuQuiescentState all finish before the writer proceeds to deallocate the node.

In this simplified version, there is a fixed number of readers, \( N \), and the writer immediately synchronizes and deallocates the old node as soon as it performs an update. In the full version verified in the appendix, we allow the readers to register themselves dynamically and let the writer batch its deallocations for efficiency.

4. RCU Specification

GPS [Turon et al. 2014] lets us prove Hoare-style triples of the form:

\[ \{P\} e \{x. Q\} \]
asserting that if a thread starts with the resources described by \( P \) and executes expression \( e \), then:

- The execution of \( e \) is guaranteed to be free of memory errors (e.g. accessing deallocated data) and data races.
- If \( e \) terminates with value \( V \), then \([V/x]Q\) describes the thread’s resources afterward.

Later, we will review the logical mechanisms that GPS provides for proving these triples. For now, we describe the specification for RCU that we will prove. The full specification is shown in Figure 3. We assume some fixed predicate \( P(x) \) that we require to hold of values stored in the list. Any value inserted by the writer must satisfy this predicate, and readers are guaranteed that values they get out will also satisfy it. We use the \( \Box \) modality to denote that \( P \) is duplicable; that is, \( P \Leftrightarrow P \times P \). The RCU specification involves three predicates which are defined in terms of underlying GPS primitives but can be treated abstractly by a client: \( \text{WriterSafe}(q, L) \), \( \text{ReaderSafe}(q, H, tid) \), \( \text{SafePtr}(H, p) \).

\( \text{WriterSafe}(q, L) \) represents the permissions owned by the writer. The logical list \( L \) consists of \( q \) followed by pointers to the nodes in the list, in order. This predicate says that for the RCU structure with metadata at \( q \), the physical list contains the nodes mentioned in \( L \). We generate this permission when we create a new RCU instance, at which point \( L \) consists only of \( q \). Accessing the link field of a pointer in \( L \) just returns the next pointer in \( L \). Each of the writer’s methods consumes this permission, and returns a version where the contents of \( L \) have been modified accordingly.

\( \text{ReaderSafe}(q, H, tid) \) is the analogous permission for readers. From the perspective of the client code, \( H \) is completely abstract: it simply represents the version of the list that the \( tid \)-th reader sees. \( \text{SafePtr}(q, H, p) \) means that \( p \) is a pointer to a properly initialized node. The specification for \( \text{rcuReadStart} \) says that it always returns a \( \text{SafePtr} \). As the reader inspects the list using \( \text{rcuReadNext} \), \( p \) must be a non-null \( \text{SafePtr} \), and when the call returns, it returns another \( \text{SafePtr} \).

When the reader calls \( \text{rcuQuiescentState}(q) \), it gives up its current \( \text{ReaderSafe}(q, H, tid) \) and in return it receives \( \text{ReaderSafe}(q, H', tid) \) for some fresh \( H' \). This makes any previous \( \text{SafePtr} \) facts unusable, and forces the reader to start again at the head of the list by getting a new \( \text{SafePtr} \) from \( \text{rcuReadStart} \).

5. GPS

In this section, we briefly review some of the key mechanisms in GPS that we will use in verifying our RCU implementation. We then illustrate their use on the simple message-passing example from §2. Although the example is contrived, its verification closely mirrors the structure of the RCU verification, and it shows off all the features of GPS working in tandem. It is thus quite useful as a warm-up for the main attraction.

5.1 Key Features of GPS

The four key features of GPS are as follows.

**Ownership of non-atomics:** The assertion \( x \leftrightarrow v \) says that \( x \) is a non-atomic location pointing to the value \( v \). This assertion is precisely the standard points-to assertion of separation logic [Reynolds 2002]: whoever asserts \( x \leftrightarrow v \) is the exclusive “owner” of \( x \), and has the freedom to read and write it arbitrarily. Here, we also extend GPS slightly to support fractional permissions [Borntat et al. 2005] on non-atomic locations: \( x \xrightarrow{k} v \) (where \( 0 \leq k < 1 \)) denotes ownership of a \( k \) fraction of \( x \), which means the ability to read \( x \) but not write it. The initial (full) owner of \( x \) may thus split up her ownership assertion into fractional pieces to be given out to readers, and then later on collect those pieces to reconstitute the full permission so that she can update and/or deallocate \( x \). Crucially, though, with neither full nor fractional ownership is it possible for one thread to read \( x \) at
the same time another may be writing it; thus, we guarantee absence of data races on non-atomics.

**Protocols for message passing via atomics:** Unlike non-atomics, atomic locations are meant to be read and written simultaneously. We therefore cannot make any stable assertions about the precise contents of an atomic location, but we can assert something about how those contents are permitted to evolve over time. We call such an assertion a protocol assertion, $[x : s \tau]$. It asserts two things. First, it says that $x$ is governed by the protocol $\tau$. This protocol consists of an ordered set of logical states $S$ that $x$ can be in, together with an interpretation function $\tau(s, v)$ that says what assertion must hold when $x$ is in logical state $s \in S$ and stores value $v$. Second, the protocol assertion says that $x$ is at least in state $s$ of its protocol. This assertion is duplicable, and may thus be shared freely between threads, because GPS requires writes to $x$ to always advance the state of its protocol—so once $x$ is at least in state $s$, it will remain so forever.

Through their interpretation functions, protocols offer a way for threads to pass messages to each other. Specifically, suppose two threads both know $[x : s \tau]$. When one of the threads writes $v$ to $x$, it must be able to prove that $\tau(s', v)$ holds for some future state $s'$ of $s$. Subsequently, when the other thread performs a read on $x$, observing value $v$, it will learn that there is some future state $s'$ of $s$ such that $\tau(s', v)$ holds. The protocol has thus served to transmit the knowledge of $3s' \supseteq s$, $\tau(s', v)$ from one thread to the other.

**Exchanges for ownership transfer:** While protocols support the transfer of knowledge (i.e., duplicable facts) between threads, exchanges support the transfer of exclusive ownership of resources between them. This will be very important when verifying our message-passing example (see §5.2 below), wherein we want to pass exclusive ownership of $x \mapsto 37$ back and forth between the two threads.

The exchange mechanism is very simple. Suppose $P$ and $Q$ are assertions such that $P \land Q \Rightarrow \text{false}$ and $Q \land P \Rightarrow \text{false}$, i.e., they denote exclusive ownership, so two threads cannot assert $P$ simultaneously (and likewise for $Q$). We write $\sigma : P \leftrightarrow Q$ to say that $\sigma$ is the name of an exchange between $P$ and $Q$, and we write $\text{exch}(\sigma)$ to represent the assertion that the exchange $\sigma$ has been created. The idea is that $\sigma$, once created, represents an invariant governing some shared state, which asserts that that shared state either satisfies $P$ or it satisfies $Q$. Once created, the $\sigma$ invariant is enforced permanently, and thus the assertion $\text{exch}(\sigma)$ is duplicable knowledge that can be freely shared amongst threads.

To see how exchanges support ownership transfer, suppose thread 1 owns $P$, thread 2 owns $Q$, and thread 1 wishes to transfer ownership of $P$ to thread 2. Thread 1 can create the exchange $\sigma$ by giving up ownership of $P$ to the exchange, thereby learning $\text{exch}(\sigma)$ in return. It may then use release-acquire message passing (as described above) to inform thread 2 of the knowledge that $\sigma$ exists. Since thread 2 owns $Q$, it can then give up $Q$ in exchange for $P$. These logical ownership transfers are summarized as follows:

$$ (P \lor Q) \Rightarrow \text{exch}(\sigma) $$

$$ P \land \text{exch}(\sigma) \Rightarrow Q \quad Q \land \text{exch}(\sigma) \Rightarrow P $$

Note that the assumption that $P$ and $Q$ are exclusive (non-duplicable) is essential in order to ensure that there is a unique recipient of the ownership transfer. For instance, if $Q$ were some duplicable fact, then multiple threads would be able to exchange $Q$ for $P$, which would result (unsoundly) in multiple threads simultaneously owning $P$.

**Ghost PCMs for encoding auxiliary state:** Ghost (or auxiliary) state is a ubiquitous mechanism in program logics, enabling the verifier to record and manipulate additional logical state beyond the physical state manipulated by the program itself. GPS supports a very general notion of ghost state in the form of user-defined partial commutative monoids (PCMs). Given a PCM $\mu$ and an element $t \in \mu$ we write $[\gamma : t \mu]$ to say that a thread owns element $t$ from instance $\gamma$ of the monoid $\mu$. Then, we can split or combine ghost state elements as follows:

$$ [x : s \tau] \leftrightarrow [\gamma : t \mu] \Rightarrow \mu(s) \cdot [\gamma : t \mu] $$

In particular, if $t \cdot \mu t'$ is not defined (which is possible since the monoid is partial), then $[\gamma : t \mu] \cdot [\gamma : t' \mu] \Rightarrow \text{false}$.

Recent work has shown that PCMs are remarkably expressive [Jung et al. 2015]. In this paper, we focus on two relatively simple types of PCMs that are relevant to the RCU proof: permission tokens and history snapshots.

Permission tokens represent capabilities to perform certain operations. We will use them, for instance, to represent the permissions to make certain steps in a protocol, or the permission to access certain exchanges. Permission tokens are defined by the monoid $\text{Tok}$, which has two elements $\circ$ and $\epsilon$, with $\epsilon$ as identity and $\circ \circ$ undefined. The key property of a permission token is that it is non-duplicable $[\gamma : t \circ \text{Tok}] \Rightarrow \text{false}$ and thus represents an exclusive capability. We discuss history snapshots in §6.1.

### 5.2 Verifying the Message-Passing Example

To show how the above mechanisms work together, let us return to the first example in §2. Our verification, which we describe here at a high (but still detailed) level, guarantees two things: (a) the postcondition, namely that, once the threads terminate, $y$ points to 25 or 37, and (b) that the code is “safe”, meaning that $x$ will not be accessed after it is freed and that there are no data races on the non-atomic $x$ and $y$.

```language=java
[x]_na := 37;  
[y]_na := 25;  

if ((y]_na == 25) {
    [y]_na := [x]_na;
}

[y]_na := 25;

while (([y]_acq != 1) {
    /* spin */
}

free(x);

/* postcond: y \mapsto 37 \lor y \mapsto 25 */
```

---

2 The original version of GPS featured a slightly more limited primitive called escrows. Exchanges generalize escrows to support bidirectional transfer.
We walk through the proof now step by step. These steps are illustrated pictorially in Figure 5, and they involve protocols and exchanges that are defined formally in Figure 4.

At the start of the proof, we associate the flags \( \ell f \) and \( r f \) with the left and right flag protocols \( LFP \) and \( RFP \), respectively (explained below). We also create the left and right permission tokens, \([ltok: []\{Tok\}]\) and \([rtok: []\{Tok\}]\), and give the left and right threads exclusive ownership of their respective tokens.

**Step 1** (Fig. 5a): The left thread first sets \( x \) to 37. It then wants to transfer ownership of \( x \) to the right thread. To do so, it creates the exchange \( LX \). By giving up ownership of \( x \) to \( LX \), it gains the knowledge \( \text{exch}(LX) \) that \( LX \) exists.

**Step 2** (Fig. 5a): The left thread now wants to send its knowledge of \( \text{exch}(LX) \) to the right thread by setting its flag, \( \ell f \), to 1. To reason about this, we use the left flag protocol \( LFP \). This protocol asserts that \( x \) is initially 0, and that it may be set to 1 but can never be set back to 0 again. It also asserts that when \( \ell f \) is set to 1, it must be the case that \( \text{exch}(LX) \) holds. Since the left thread knows \( \text{exch}(LX) \), it is free to update \( \ell f \) to 1 (updating the logical state \( s \) of \( \ell f \)'s \( LFP \) protocol to 1 as well).

**Step 3** (Fig. 5b): The right thread may or may not observe that \( \ell f \) has been set to 1. In case it does not observe it, this and the next step are skipped. In case it does observe it, it learns that the \( LFP \) protocol must be in the 1 state, and hence it learns that \( LX \) exists. It then uses \( LX \) to exchange its own permission token, \( rtok \), for ownership of \( x \) to 37. Now that it owns \( x \), it can safely read it and be sure that it will see the value 37.

**Step 4** (Fig. 5b): The right thread now wants to transfer ownership of \( x \) back to the left thread. To achieve this, its first step is to perform the reverse trade on \( LX \), putting ownership of \( x \) back under control of \( LX \) in return for its permission token \( rtok \). **Note**: at this point, regardless of whether Steps 3 and 4 were performed or not, \( y \) points to either 25 or 37, as desired.

**Step 5** (Fig. 5c): The right thread next creates the exchange \( RX \) by transferring its permission token \( rtok \) into the exchange. In doing so, it learns \( \text{exch}(RX) \).

**Step 6** (Fig. 5c): The right thread now wants to send its knowledge of \( \text{exch}(RX) \) to the left thread by setting its flag, \( rf \), to 1. To reason about this, we use a right flag protocol, \( RFP \), that is very similar to the left flag protocol, \( LFP \), the only difference being that in state 1, \( RFP \) asserts \( \text{exch}(RX) \) (rather than \( \text{exch}(LX) \)). The right thread may thus set \( rf \) to 1 because it knows \( RX \) exists.

**Step 7** (Fig. 5d): The left thread loops until it observes that \( rf \) has been set to 1. Once it observes this, it knows that the \( RFP \) protocol must be in state 1 and thus that \( RX \) exists. It then uses \( RX \) to exchange its own permission token, \( ltok \), for the right permission token, \( rtok \).

**Step 8** (Fig. 5d): Finally, the left thread uses its original \( LX \) exchange to trade the right permission token, \( rtok \), for ownership of \( x \). It now knows that it has exclusive ownership of \( x \) and may therefore safely deallocate it!

Suppose that in addition we wanted to deallocate the atomic locations \( \ell f \) and \( rf \) at the end of the program. Using the original protocol mechanism of GPS, it is not possible to deallocate atomic locations because the assertion \( \ell : s \{T \} \) is a sufficient precondition for performing release writes and acquire reads on \( \ell \). That is, since protocol assertions are duplicable, it would never be safe to deallocate \( \ell \): some thread could still have a copy of its protocol assertion and subsequently attempt to read or write it.

We can extend GPS with support for deallocation of atomic locations by introducing abstract locations. Instead of associating atomic locations directly with a protocol, we instead associate them with abstract locations. It is these
abstract locations which now have a protocol attached. If \( \varphi \) is an abstract location, then we write \( \ell \rightarrow \varphi \) and \( \varphi : s \vdash T \) to say that physical atomic location \( \ell \) is associated with \( \varphi \), and \( \varphi \) is at least in state \( s \) on protocol \( T \). With these two facts, a thread can read from \( \ell \) and thereby observe a state that \( \varphi \) may have moved to, or it can write to \( \ell \) and update the state of \( \varphi \).

Under this extension, although \( \varphi : s \vdash T \) remains duplicable, \( \ell \rightarrow \varphi \) must be exclusively owned. Just as with \( x \rightarrow v \), we can split this \( \ell \rightarrow \varphi \) into fractional pieces. However, unlike for non-atomics, a fractional piece suffices for both reading and writing. The full permission \( \ell \rightarrow \varphi \) is required only in order to deallocate \( \ell \) or to change the abstract location that \( \ell \) is associated with. A thread can safely perform these operations if it has the full permission, because it knows that no other thread could be reading to or writing to the location.

6. RCU Proof Overview

In §5, we saw how protocols and exchanges could be used to implement message passing and ownership transfer. We now use these mechanisms to formalize the intuitive explanation we gave for RCU in §3. There are two important ownership transfers involved in RCU: the writer transfers fractional ownership of nodes to readers when the nodes are added to the list, and readers transfer the fractional ownership of deleted nodes back to the writer during synchronization.

We first explain at a high level how protocols and exchanges are used in the steps of the algorithm. Figure 6 shows the message passing and ownership transfer involved in the release-acquire pairs in terms of the predicates and exchanges used in the proof. Figure 7 contains the formal definitions.

Note that these definitions refer to various pieces of ghost state (e.g., tokens) projected from \( \Gamma \). This \( \Gamma \) is just a record collecting together all the state associated with a particular instance of an RCU linked list.

Step 1 (Fig. 6a): After initializing a new node with value \( v \) at physical location \( x \), the writer will own \( x + \text{data} \leftrightarrow v \) and \( x + \text{link} \rightarrow i \), for some logical location \( i \). It splits these permissions into pieces for each reader. For each reader thread \( tid < N \), it creates an exchange \( \text{PermX}(\Gamma, x, tid, i) \). On one side of this exchange are the fractional permissions of \( x + \text{data} \) and \( x + \text{link} \) for reader \( tid \). The other side of this exchange is the \( (tid, i) \) token from the \( \Gamma \cdot \text{rxtok} \) ghost state. Reader \( tid \) begins by owning all of the \( \{tid\} \times \mathbb{N} \) of these tokens. (Compare this setup with the LX exchange that was created in Step 1 of the message-passing example in §5.2.)

Step 2 (Fig. 6b): We set up a protocol LLP on the \text{link} field, which formalizes Release-Acquire Pair 1 from §3. When the writer updates the \text{link} field of the parent \( p \) to point to \( x \), it will store the knowledge that each of the \( \text{PermX}(\Gamma, x, tid, i) \) exchanges have been created. (Compare this with the use of LLP in Step 2 in §5.2.)

Step 3 (Fig. 6c): When reader \( tid \) reads \( p + \text{link} \), they will learn about \( \text{PermX}(\Gamma, x, tid, i) \), and use its \( \Gamma \cdot \text{rxtok} \) token to get access to \( x \)'s fields.

Step 4 (Fig. 6d): When the reader is done with the node, it uses \( \text{PermX} \) in the opposite direction to get its \( \Gamma \cdot \text{rxtok} \) token back. (Compare this and the previous step with Steps 3 and 4 in §5.2.)
Step 5 (Fig. 6e): Now, suppose later on the writer has deleted the node at location $x$, where $x + 1 \lnk$ is tied to some logical location $i$, and now wants to deallocate it. To do so, it increments $\wco$, which in turn is governed by protocol WCP (formalizing Release-Acquire Pair 2 from §3). It stores $\text{SnapshotValid}(H)$, which asserts that the abstract locations for the $\lnk$ fields of the nodes in the list are in fact in the state suggested by the "history snapshot" $H$. We explain history snapshots in §6.1, but intuitively, they represent the history of updates to the list, and $H$ here is the most up-to-date history. In particular, since $x$ has been deleted from the list, $H$ here will mark $x$ as a dead node. When reader $\text{tid}$ sees the updated writer's counter, it infers that the writer wants to deallocate $x$ because $x$ is marked as dead in $H$.

Step 6 (Fig. 6f): For each of the nodes requested by the writer for deallocation, reader $\text{tid}$ looks at the abstract location $i$ of that node's $\lnk$ field, and creates a $\text{ModX}(\Gamma, \text{tid}, i)$ exchange. One side of this exchange is the $(\text{tid}, i)$-th $\Gamma{\cdot rxtok}$. The other side is a corresponding $(\text{tid}, i)$ token from $\Gamma{\cdot wxtok}$. Here, $\Gamma{\cdot wxtok}$ is a set of tokens that the writer starts with, which it uses to retrieve the reader's tokens. (Compare with the creation of the RX exchange in Step 5 in §5.2.)

Step 7 (Fig. 6f): The reader transmits its knowledge of the existence of these $\text{ModX}$ exchanges by updating its counter, $\text{rcounters}+\text{tid}$. This counter is in turn governed by protocol RCP$(\Gamma, \text{tid})$, formalizing Release-Acquire Pair 3 from §3. (Compare with the use of protocol RFP in Step 6 in §5.2.)

Step 8 (Fig. 6g): As the writer sees the updated $\text{rcounters}+\text{tid}$ fields, it uses each $\text{ModX}(\text{tid}, i)$ it learns about to exchange its own $(\text{tid}, i)$-th $\Gamma{\cdot wxtok}$ token for the corresponding $(\text{tid}, i)$-th $\Gamma{\cdot rxtok}$ token. (Compare with the token exchange that occurs in Step 7 in §5.2.)

Step 9 (Fig. 6g): Finally, it uses these $\Gamma{\cdot rxtok}$ tokens with the $\text{PermX}(\Gamma, x, \text{tid}, i)$ exchange to get back all the fractional permissions for $x$. After it has done this for for every reader's token, it will have collected the full permission for $x$, and it may deallocate the node. (Compare with the final Step 8 in §5.2.)

In the remainder of this section, we first present some more detail about the ghost state constructions needed in the proof (§6.1). We then explain the definitions of the abstract predicates $\text{ReaderSafe}$ and $\text{SafePtr}$ (from the spec in Figure 3) and sketch why the reader specifications are correct (§6.2). We conclude with a brief discussion of the extensions to our basic RCU implementation that are supported by our full verification (§6.3). The definition of $\text{WriterSafe}$ and the full Hoare-style proofs are given in the appendix.

### 6.1 Ghost State

Our proof uses ghost state in three ways: (1) as permission tokens for exchanges, (2) to control the progress of protocols, and (3) to track the state (and more generally the history) of the linked list.

#### Exchange tokens

We have the $\Gamma{\cdot rxtok}$ and $\Gamma{\cdot wxtok}$ tokens for the $\text{PermX}$ and $\text{ModX}$ exchanges. As we want a fresh token for each thread and for each location, we take the monoid to be the powerset of $\{0, \ldots, N-1\} \times \mathbb{N}$ with disjoint union as composition. Reader $i$ starts with the set $\{i\} \times \mathbb{N}$ of $\Gamma{\cdot rxtok}$. Meanwhile, the writer starts with all of the $\Gamma{\cdot wxtok}$.

#### Protocol state tokens

Each thread has a counter which only it is allowed to modify. The way we can ensure that only one thread may write to a location $l$ is to give the thread a
set of tokens corresponding to the states in the protocol. The interpretation function for the protocol then requires that to move to state \( s \), the thread must give up the token which matches \( s \). The thread begins with all of these tokens and deposits one each time it updates the state of the list. It knows that no other thread could have concurrently updated the state of \( l \), because it owns the unique token needed for the update.

For RCU, we use the powerset of \( \{0, \ldots, N\} \times N \) monoid for these tokens. The instance of this monoid is called \( \Gamma.ctok \) (for “counter token”). Reader \( i \) starts with \( \{i\} \times N \), and the writer starts with \( \{N\} \times N \). Then, we can set up the WCP and RCP protocols for the counters so that to update their values to \( v \), a thread needs to deposit counter token \( (i, v) \).

**History snapshots** As part of Release-Acquire Pair 2, the writer needs to inform the readers that the node it wants to deallocate is no longer reachable. To do this, we record the history of the list as a piece of ghost state \( H \), which is a list of abstract actions taken by the writer. Actions are of the form \( \text{upd}(i, i', v) \) or \( \text{del}(i) \), where \( i \in N \) and \( i' \in N \cup \{\text{null}\} \). The former represents updating the link pointer for node \( i \) to point to node \( i' \) (if \( i' \in N \)) or null (if \( i' = \text{null} \)). The latter indicates the writer’s intention to deallocate node \( i \).

Given a history \( H \) and an abstract location \( i \), we can consider the subhistory of \( H \) containing only actions of the form \( \text{upd}(i, i') \) or \( \text{del}(i) \). We call this the subhistory of \( H \) restricted to \( i \), written \( H_i \). For convenience, we treat \( H \) as a partial function, writing \( H(i) = T \) if \( \text{del}(i) \in H \), and \( H(i) = i_n \ldots i_1 \) if \( H_i = \text{upd}(i, i_1) \ldots \text{upd}(i, i_n) \).

If the first action in \( H \) is either \( \text{upd}(i, -) \) or \( \text{del}(i) \) we say that the base of \( H \), written \( \text{base}(H) \), is \( i \). The base \( H \) is the abstract location corresponding to \( q + \text{link} \), the pointer to the head of the list. We define \( \text{dead}(H) \) to be the set of all \( i \) such that \( H(i) = T \) and \( \text{live}(H) \triangleq \text{dom}(H) \setminus \text{dead}(H) \). We restrict the set of histories to well-formed ones where no node ever points to a dead node or an uninitialized node.

Histories can be ordered by saying that \( H_1 \preceq H_2 \) if \( H_1 \) is a prefix of \( H_2 \). This ordering has the following useful properties:

- If \( H_1 \preceq H_2 \) then \( \text{dead}(H_1) \subseteq \text{dead}(H_2) \).
- If \( H_1 \preceq H_2 \), then \( \text{dom}(H_1) \subseteq \text{dom}(H_2) \). In addition if \( i \in \text{live}(H_1) \) and \( i \in \text{live}(H_2) \), then there exists \( L \) such that \( H_2(i) = L \cdot H_1(i) \).

We are now ready to define history snapshots, that is, duplicable observations about the history of the list. History snapshots will be modeled by a PCM with elements of the form \( \text{Master}(H) \) and \( \text{Snapshot}(H) \) for each well-formed history \( H \). Then the monoid operation \( \cdot \) is defined by:

\[
\text{Snapshot}(H) \cdot \mu \cdot \text{Master}(H') \triangleq \text{Master}(H') \text{ iff } H \leq H' \\
\text{Master}(H) \cdot \mu \cdot \text{Snapshot}(H') \triangleq \text{Master}(H) \text{ iff } H' \leq H \\
\text{Snapshot}(H) \cdot \mu \cdot \text{Snapshot}(H') \triangleq \text{Snapshot}(\text{max}(H, H')) \\
\text{(if this exists)}
\]

Note that since \( \text{Master}(H) \) is not composable with any other \( \text{Master}(H') \), it means that whichever thread owns \( \text{Master}(H) \) owns the authoritative state of the history.

In our RCU proof, \( \Gamma \)’s history is an instance of this snapshot monoid. In our case, the writer is the thread who owns the authoritative \( \text{Master}(H) \) and this puts her in a privileged position. First of all, she is allowed to update the state of the history monoid to \( \text{Master}(H') \) so long as \( H' \geq H \), i.e., so long as she only extends the history with new actions that do not invalidate any existing snapshots. (Formally, this is enabled by GPS’s “frame-preserving ghost update rule”.) Second, she can copy off duplicate snapshots of this master, which she will then store in her counter and the \( \text{link} \) fields of the nodes. Since these snapshots are duplicable, they can be passed to readers as part of the protocol for Release-Acquire Pair 2. If a reader has \( \text{Snapshot}(H) \), then it knows this \( H \) is a lower bound on the state of the master history. In particular, if a node is in \( \text{dead}(H) \), the reader knows that it must always be in the dead set of the master copy, too.

6.2 Reader Abstract Predicates

We now give the parts of the definitions of ReaderSafe and SafePtr that are relevant for accessing nodes in the list:

\[
\text{ReaderSafe}(q, H, tid) \triangleq \exists \Gamma. q = q \ast \text{SnapshotValid}(\Gamma, H) \\
\ast \Gamma.rxtok : \{tid\} \times (N \setminus \text{dead}(H)) \ast (\text{counter resources}) \\
\text{SafePtr}(q, H, p) \triangleq \exists \Gamma. q = q \ast \text{SnapshotValid}(\Gamma, H) \\
\ast (p \neq 0) \Rightarrow \exists i. i \notin \text{dead}(H) \\
\ast \forall tid < N. \text{exch}(\text{PermX}(\Gamma, p, tid, i)))
\]

The ReaderSafe \((q, H, tid)\) predicate asserts that \( H \) is a valid snapshot and contains all the \( \Gamma.ctok \) tokens for thread \( tid \) except for the nodes that are dead in \( H \). From this definition, it is clear how we lose ReaderSafe\((q, H, tid)\) during \text{rcuQuiescentState}, and get back ReaderSafe\((q, H', tid)\) for some \( H' \). During this function, the reader will give up some of its \( \Gamma.ctok \) by creating exchanges. However, it learns that \( \text{SnapshotValid}(H') \) for some new \( H' \), and only gives up tokens in \( \text{dead}(H') \).

Then, SafePtr \((q, H, p)\) just says that if \( p \) is non-null, then there is \( \text{PermX}(\Gamma, p, tid, i) \) for some \( i \notin \text{dead}(H) \). In this case, if the reader has both of these facts, then it must have \( \Gamma.ctok : \{tid, i\} \), so it can use this exchange to get access to the node located at \( p \). From this we can see why the precondition for \text{rcuReadNext} is sufficient.

The postcondition for \text{rcuReadNext} requires us to prove that when the reader does an acquire read on \( p + \text{link} \) (line 8) and gets some value \( p' \), that SafePtr \((q, H, p')\) is also true. Now, we know from \( \text{PermX}(\Gamma, p, tid, i) \) that \( p + \text{link} \) is associated with abstract location \( i \), and from \( \text{SnapshotValid} \), we gather that \( i \) is at least in state \( H(i) \) (if \( i \in \text{dom}(H) \)). When we read \( p + \text{link} \), the protocol guarantees that if \( p' \neq 0 \), then there exists some snapshot of an \( H' \) and \( i' \) such that \( \text{hd}(H'(i)) = i' \). Moreover, if \( i \in \text{dom}(H) \), then
we know that $H(i) \leq H'(i)$, since $H'(i)$ corresponds to the possibly later state of $i$. In addition, we also learn that are 
PerM($\Gamma, p', tid, i'$) for each $tid$. To show SafePtr($q, H, p'$), it suffices to show that $i' \not\in \text{dead}(H)$. This follows from the following lemma on histories:

**Lemma.** Let $H$ and $H'$ be comparable histories such that $i \not\in \text{dead}(H)$ and $\text{hd}(H'(i)) = i'$. Then if $H(i) \leq H'(i)$ or $i \not\in \text{dom}(H)$, then $i' \not\in \text{dead}(H)$.

**Proof.** The proof proceeds by cases on whether $i \in \text{dom}(H)$ and whether $H \leq H'$ or $H' \leq H$. In each subcase, the argument follows from the fact that the dead set and domain of a history are monotonic in its length. 

### 6.3 Extensions to the Basic RCU Implementation

The full version of RCU verified in the appendix contains two additional features:

- the writer batches together several node deallocations, and
- readers dynamically register themselves.

Supporting batched deallocation is straightforward. During rcuNodeUpdate, the writer adds the old node to a deallocation queue. Then, when it wants to deallocate the stack, it performs rcuSynchronize, and gets the reader tokens for all nodes in the stack at once.

For dynamic registration, the RCU metadata contains an additional field, $q + \text{numreaders}$, which is a counter storing the number of readers. To register, a reader does a fetch-and-increment on this field to get their $tid$. During rcuSynchronize, the writer reads $q + \text{numreaders}$ and only examines the $q + \text{rcounters}$ entries for registered readers. For the proof, we have a protocol on $q + \text{numreaders}$ that initially contains ReaderSafe($q, H, tid$) for all $tid$. During the fetch-and-increment, the reader takes out the ReaderSafe($q, H, tid$) for the $tid$ it gets assigned. Similarly, during rcuSynchronize, when the writer wants to deallocate the node at logical location $i$, it takes out $\Gamma\cdot\text{rxtok}(tid, i)$ for all readers who have not yet registered.

### 7. Related Work

**User-space RCU.** Our implementation of RCU is based on that of Desnoyers et al. [2012], who describe a number of RCU implementations, of which QSBR is one that provides highly optimized performance. Their implementation uses memory barriers rather than C11 concurrency primitives. In addition, their implementation of QSBR allows readers to go “offline” for extended periods by setting their counter field to $-1$. When the writer performs rcuSynchronize, it then checks that each reader’s counter either matches its own or is $-1$. Later, the reader can go back online by copying the writer’s counter value again. We left a proper treatment of this extension for future work because it requires a combination of weak-memory primitives and stronger synchronization operations (SC fences), for which currently no adequate verification techniques exist.

**Consume reads.** The C11 memory model also contains consume reads, which are weaker than acquire reads and cheaper to implement efficiently on the Power and ARM architectures. With acquire reads, *everything* following the read is guaranteed to happen after the things preceding the matching release write. However, with consume reads, only the things that have a data dependency on the value read are guaranteed to happen after. For example,

\[
\begin{align*}
[x]_{wa} & := 25; \\
[y]_{wa} & := 37; \\
[m]_{wa} & := x; \\
[n]_{wa} & := [p]_{wa}; \\
b]_{wa} & := [y]_{wa}; \\
\end{align*}
\]

In this example, the right thread’s read through the pointer $p$ is guaranteed to happen after the write that initialized $x$, because this access depends on the value from the consume read of $m$. In contrast, the access to $y$ is racy, because it does not have a data dependency, only a control dependency. This ordering is sufficient for RCU if the reader does consume reads on the $\text{lnk}$ field, because the accesses to the fields of the node will depend on the pointer it reads. In fact, RCU was a primary motivation for including consume reads in the C11 standard. However, the standard may be revised because the current rules for data dependency tracking are too complicated, and most compilers treat consume reads as acquire reads [McKenney et al. 2014]. Once these revisions are finalized, we believe it should be possible to extend GPS with support for consume reads through a modality that tracks dependencies.

**Other RCU proofs.** Gotsman et al. [2013] verify an RCU-based non-blocking stack implementation under a sequentially consistent memory model. Their RCU synchronization procedure is closer to exclusively using the offline/online feature of QSBR described above. They formalize the concept of a *grace period*, which is often used to informally explain RCU. A grace period is the length of time from when a node becomes unreachable until no readers are accessing it any longer. They show that this concept can be used to structure the proofs of related memory management techniques such as hazard pointers and epoch-based reclamation. Their proof uses a concurrent separation logic extended with temporal operators to make statements about the grace period. It would be interesting to try to add such temporal operators to a logic like GPS and see if a proof based on grace periods can be formalized in the setting of weak memory, but as we have shown, one can verify an RCU implementation even without them.

**Relaxed Separation Logic (RSL).** One reason perhaps why there has been no prior work on formally verifying RCU in a weak-memory setting is that program logics for weak-memory concurrency have only begun appearing very recently. For instance, it was only in 2013 that Vafeiadis and Narayan [2013] proposed RSL, the first program logic for the
C11 memory model. RSL is a simpler logic than its sequel, GPS, but also less powerful: the GPS paper presents several examples that are beyond the scope of RSL. It is therefore in-structive to consider whether we could have verified our RCU implementation using the simpler RSL. We cannot provide a definite answer whether this is possible or not, but we believe it is rather unlikely, given how heavily our proof relies on the rely-guarantee reasoning afforded by protocols, which is not directly supported in RSL.

References


V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In POPL, ACM, 2015.
A. Language

A.1 Syntax

\[
\begin{align*}
\text{Val} & \quad V ::= \ n \\
\text{OVal} & \quad v ::= \ x \mid V \\
\text{Exp} & \quad e ::= \ v \mid v + v \mid v = v \mid v \mod v \mid \text{let } x = e \text{ in } e \mid \text{repeat } e \text{ end} \mid \text{fork } e \\
& \quad \quad \quad \mid \text{if } v \text{ then } e \text{ else } e \mid \text{alloc}(n) \mid [v]_O \mid [v]_O := v \mid \text{CAS}(v, v, v) \mid \text{FAI}(v)
\end{align*}
\]

\[
\begin{align*}
\text{OrderAnn} & \quad O ::= \ at \mid na \\
\text{EvalCtx} & \quad K ::= [[]] \mid \text{let } x = K \text{ in } e \\
\text{Action} & \quad \alpha ::= \ S \mid \text{A}(\ell..e') \mid \text{W}(\ell, V, O) \mid \text{R}(\ell, V, O) \mid \text{U}(\ell, V, V) \\
\text{ActName} & \quad a \quad \text{(an infinite set)} \\
\text{ActMap} & \quad A \in \text{ActName} \rightarrow \text{Action} \\
\text{Graph} & \quad G ::= (A, sb, mo, rf) \quad \text{sb, mo} \subseteq \text{dom}(A) \times \text{dom}(A), rf \in \text{dom}(A) \rightarrow \text{dom}(A) \\
\text{ThreadMap} & \quad T \in \mathbb{N} \rightarrow \text{(ActName} \times \text{Exp)}
\end{align*}
\]

Note: Formally, we use \textit{at} as the notation for the memory ordering for both release release writes and acquire reads. However, when displaying programs in the main body of the paper, we have written \texttt{rel} and \texttt{acq} for clarity.

A.2 Semantics

\[
\text{Event steps} \quad e \xrightarrow{\alpha} e
\]

\[
\begin{align*}
& \quad n + m \quad S \rightarrow \ k \quad k = n + m \\
& \quad n \mod m \quad S \rightarrow \ k \quad k = n \mod m \\
& \quad n = m \quad S \rightarrow \ 1 \quad n = m \\
& \quad n = m \quad S \rightarrow \ 0 \quad n \neq m \\
& \quad \text{let } x = V \text{ in } e \quad S \rightarrow \ e[V/x] \\
& \quad \text{repeat } e \text{ end} \quad S \rightarrow \ \text{let } x = e \text{ in } \text{if } x \text{ then } x \text{ else } \text{repeat } e \text{ end} \\
& \quad \text{if } V \text{ then } e_1 \text{ else } e_2 \quad S \rightarrow \ e_1 \quad V \neq 0 \\
& \quad \text{if } V \text{ then } e_1 \text{ else } e_2 \quad S \rightarrow \ e_2 \quad V = 0
\end{align*}
\]

\[
\text{Machine steps} \quad \langle T; G \rangle \rightarrow \langle T'; G' \rangle
\]

\[
\begin{align*}
& \quad e \xrightarrow{\alpha} e' \quad \text{consistentC11}(G') \\
& \quad G'.A = G.A \uplus [a' \mapsto \alpha] \\
& \quad G'.sb = G.sb \uplus (a, a') \\
& \quad G'.mo \supseteq G.mo \\
& \quad G'.rf \in \{G.rf, G.rf \uplus [a' \mapsto b]\} \\
& \quad \langle T \uplus [i \mapsto (a, e)]; G \rangle \rightarrow \langle T \uplus [i \mapsto (a', e')]; G' \rangle \\
& \quad \langle T \uplus [i \mapsto (a, K[\text{fork } e])]; G \rangle \rightarrow \langle T \uplus [i \mapsto (a, K[0])] \uplus [j \mapsto (a, e)]; G \rangle
\end{align*}
\]

The validity of these operational rules relative to the C11 standard can be found in the appendix of [Turon et al. 2014].
\[
\text{execs}(e) \triangleq \left\{ (e', G) \mid \langle [\text{start} \mapsto S], 0, 0, 0 \rangle \xrightarrow{\ast} \langle [\text{start} \mapsto (\_ e')] \cup T; G \rangle \right\}
\]

\[
\llbracket e \rrbracket \triangleq \begin{cases} \text{err} & \exists (\_, G) \in \text{execs}(e). \text{dataRace}(G) \lor \text{memoryError}(G) \\ \{ V \mid (V, \_) \in \text{execs}(e) \} & \text{otherwise} \end{cases}
\]

A.3 Memory model

A.3.1 The C11 atomic access modes

The C11 standard [ISO/IEC 9899:2011; ISO/IEC 14882:2011 2011] includes several kinds of atomic accesses: sequentially-consistent, release-acquire, release-consume, and fully relaxed. We have focused on release-acquire, because:

- Sequentially-consistent accesses are already well-understood.
- Release-consume atomics are useful only for specific architectures (PowerPC and Arm), but substantially complicate the memory model.
- Fully relaxed accesses, as formalized by Batty et al. [2011], suffer from several known problems. First, they allow out-of-thin-air reads, which the text of the standard explicitly forbids [ISO/IEC 9899:2011; ISO/IEC 14882:2011 2011]—but it is not known how to rule out these reads without also obstructing key compiler optimizations. On the other hand, even as formalized, fully relaxed access do not permit certain basic optimizations [Vafeiadis and Narayan 2013]. They also pose severe problems for compositional reasoning [Vafeiadis and Narayan 2013; Batty et al. 2013].

A.3.2 The formal C11 model

The C11 memory model we use is based on the formalization of Batty et al. [2011], as simplified by Batty et al. [2012] in the absence of release-consume atomics. We also incorporate the following simplifications introduced by Vafeiadis and Narayan [2013]:

- The sb and sw orders are not transitive; e.g., sb relates each event only to its immediate successors in program order. This simplifies both the operational semantics of the language and the semantics of GPS. Since hb is transitively closed, this has no effect on the memory model axioms.
- The “additional synchronized with” edges are incorporated into sb rather than sw, which again makes no difference for the axioms but simplifies the semantics.
- For uniformity, the sw edges include sb-related events, whereas in [Batty et al. 2012] these are ruled out. Since hb includes both sw and sb, this makes no difference to the axioms.

In addition to these simplifications, our formalization of the memory model drops release sequences, instead requiring sw edges only between immediate atomic read/write pairs. Consequently, our axioms are strictly weaker than those in e.g., Batty et al. [2011], since we require strictly fewer sw edges. GPS does not have proof rules that take advantage of release sequences, so it is sound with or without them.
A.3.3 Axioms

\[ \forall a, b. \text{mo}(a, b) \implies \exists \ell. \text{writes}(a, \ell, -), \text{writes}(b, \ell, -) \] (ConsistentMO1)

\[ \forall \ell. \text{strictTotalOrder}\{a \mid \text{writes}(a, \ell, -)\}, \text{mo} \] (ConsistentMO2)

\[ \forall b. \text{rf}(b) \neq \bot \iff \exists \ell, a. \text{writes}(a, \ell, -), \text{reads}(b, \ell, -), \text{hb}(a, b) \] (ConsistentRF1)

\[ \forall a, b. \text{rf}(b) = a \implies \exists \ell, V. \text{writes}(a, \ell, V), \text{reads}(b, \ell, V), \neg \text{hb}(b, a) \] (ConsistentRF2)

\[ \forall a, b. \text{rf}(b) = a, (\text{isNonatomic}(a) \lor \text{isNonatomic}(b)) \implies \text{hb}(a, b) \] (ConsistentRFN)

\[ \forall a, b. \text{hb}(a, b) \implies a \neq b, \neg \text{mo}(\text{rf}(b), \text{rf}(a)), \neg \text{mo}(\text{rf}(b), a), \neg \text{mo}(\text{rf}(b), a), \neg \text{mo}(b, a) \] (Coherence)

\[ \forall a, b. \text{isUpd}(a, b) = a \implies \text{mo}(a, c), \exists b. \text{mo}(a, b), \text{mo}(b, c) \] (AtomicCAS)

\[ \forall a, b \neq \ell, \ell'. A(a) = \mathbb{A}((\ell)), A(b) = \mathbb{A}((\ell')) \implies \ell = l' \] (ConsistentAlloc)

where \( \text{hb} \triangleq (\text{sb} \cup \text{sw})^+ \)

\[ \text{sw} \triangleq \{(a, b) \mid \text{rf}(a) = b, \text{isAtomic}(a), \text{isAtomic}(b)\} \]

\[ \text{reads}(a, \ell, V) \triangleq A(a) \in \{\mathbb{R}(\ell, V, -), \mathbb{U}(\ell, V, -)\} \]

\[ \text{writes}(a, \ell, V) \triangleq A(a) \in \{\mathbb{W}(\ell, V, -), \mathbb{U}(\ell, - V)\} \]

\[ \text{strictTotalOrder}(S, R) \triangleq (\exists a. R(a, a)), \]

\[ (\forall a, b, c. R(a, b), R(b, c) \implies R(a, c)), \]

\[ (\forall a, b \in S. a \neq b \implies R(a, b) \lor R(b, a)) \]

\[ \text{dataRace}(A, \text{sb}, \text{mo}, \text{rf}) \triangleq \exists \ell. \exists a \neq b \in \text{dom}(A). \]

\[ \text{accessesLoc}(a, \ell), \text{accessesLoc}(b, \ell), \text{reads}(a, - , -) \lor \text{writes}(b, - , -), \]

\[ \text{isNonatomic}(a) \lor \text{isNonatomic}(b), \neg \text{hb}(a, b), \neg \text{hb}(b, a) \]

where \( \text{hb} \triangleq (\text{sb} \cup \text{sw})^+ \)

memoryError\((A, \text{sb}, \text{mo}, \text{rf}) \triangleq \exists \ell. \exists b \in \text{dom}(A). \)

\[ \text{accessesLoc}(a, \ell), \text{accessesLoc}(b, \ell), \]

\[ \exists a \in \text{dom}(A). A(a) = \mathbb{A}(\ell), \ell \in \ell', \text{hb}(a, b) \]

where \( \text{hb} \triangleq (\text{sb} \cup \text{sw})^+ \)

B. Logic

B.1 Parameters

We assume:

- The following domains, with associated metavariables:

  \[ \text{s} \in \text{State} \quad (\text{a set}) \quad \tau \in \text{ProtTy} \quad (\text{a set}) \quad \sigma \in \text{ExchangeTy} \quad (\text{a set}) \quad \mu \in \text{PCMTy} \quad (\text{a set}) \]

- For each \( \mu \), a partial commutative monoid \( \llbracket \mu \rrbracket \) with unit \( \varepsilon_\mu \), multiplication \( \cdot_\mu \), and a homomorphism \( \vert - \vert : \llbracket \mu \rrbracket \to \llbracket \mu \rrbracket \) such that \( |m| = |m| \cdot_\mu |m|, |m| \leq m \) and \( |m| = m \iff m \cdot_\mu m = m \).

- For each \( \tau \) a partial order \( \subseteq_\tau \subseteq \text{State} \times \text{State} \).

- The following interpretation functions for protocols and exchanges

  \[ \text{interp}(\tau) \in \text{State} \times \text{Val} \to \text{Prop} \quad \text{interp}(\sigma) \in \text{Prop} \times \text{Prop} \]

  where if \( \text{interp}(\sigma) = (\mathcal{P}, \mathcal{P}') \) then \( \mathcal{P} * \mathcal{P} = \emptyset \) and \( \mathcal{P}' * \mathcal{P}' = \emptyset \).

- A syntax of states and PCM terms, with appropriate sorting rules, as part of the term syntax given below.

B.2 Syntax

\[ \text{Sort} \quad \theta ::= \text{Val} \mid \text{State} \mid \text{PCM}_\mu \]

\[ \text{Var} \quad X ::= \ell \mid x \mid s \]

\[ \text{Term} \quad t ::= X \mid n \mid \varepsilon_\mu \mid t \cdot_\mu t \mid \cdots \]

\[ \text{Proposition} \quad P ::= t = t \mid P \land P \mid P \lor P \mid P \implies P \mid \forall X: \theta. P \mid \exists X: \theta. P \mid \Box P \]

\[ \mid P * P \mid \text{uninit}(t) \mid t \leftrightarrow t \mid \{ t : \ell \} \mid \{ t \} \mid \{ t \} \mid \text{exch}(\sigma) \]

15 2015/2/3
B.3 Proof theory

B.3.1 Necessitation

\[ \Box P \Rightarrow P \quad \Box P \Rightarrow \Box \Box P \quad \Box P \land Q \iff \Box P \land \Box Q \quad t = t' \Rightarrow \Box t = t' \quad \ell : t \models t \Rightarrow \Box \ell : t \models t \]

\[ \text{exch}(\sigma) \Rightarrow \Box \text{exch}(\sigma) \]

B.3.2 Separation

\[ \{ \gamma : t \models \mu \} \Rightarrow \Box \{ \gamma : t \models \mu \} \]

\[ \ell \not\sim k, \varphi \not\sim \ell \not\sim k', \phi \not\sim \ell \not\sim k, \varphi \wedge k \leq 1 \wedge \varphi = \varphi' \]

\[ \ell \not\sim k, \varphi \not\sim \ell \not\sim k', \varphi \Rightarrow \ell \not\sim k, \varphi \not\sim \ell \not\sim k' \varphi \Rightarrow \ell = \ell' \]

B.3.3 Ghost moves

\[ P \Rightarrow Q \]

\[ P \Rightarrow Q \quad \quad Q \Rightarrow R \quad \quad \sigma : P \Rightarrow Q \\
Q \Rightarrow \text{exch}(\sigma) \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quan
Nonatomics

\{ \text{uninit}(\ell) \lor \ell \leftrightarrow - \} [\ell]_{\text{na}} := v \{ \ell \leftrightarrow v \} \quad \left\{ \ell \xrightarrow{k} v \right\} [\ell]_{\text{na}} \left\{ x. x = v * \ell \xrightarrow{k} v \right\} \quad \{ \text{uninit}(\ell) \lor \ell \leftrightarrow - \} \text{free}(\ell) \{ \text{true} \}

Structural rules

\[ \frac{P' \Rightarrow P}{P' \text{ e } \{ x. Q \}} \quad \forall x. Q \Rightarrow Q' \quad \frac{P \text{ e } \{ x. Q \}}{P \text{ e } \{ x. Q' \}} \quad \frac{P \text{ e } \{ x. Q \}}{P \text{ e } \{ x. Q * R \}} \]

Axioms for pure reductions

\begin{align*}
\{ \text{true} \} & \quad v \quad \{ x. x = v \} \\
\{ \text{true} \} & \quad v + v' \quad \{ x. x = v + v' \} \\
\{ \text{true} \} & \quad v == v' \quad \{ x. x = 1 \leftrightarrow v = v' \} \\
\{ P \} & \quad \forall x. \{ Q \} e' \{ y. R \} \\
\{ P \} & \quad \text{let } x = e \text{ in } e' \{ y. R \} \\
\{ P \} & \quad \text{fork } e \text{ true} \quad \{ P \} \text{ e } \{ x. (x = 0 \ast P) \lor (x \neq 0 \ast Q) \} \\
\{ P \} & \quad \text{repeat } e \text{ end } \{ x. Q \} \\
\end{align*}

C. RCU

We verify a simple version of quiescent-state-based reclamation (QSBR) RCU based on the implementation described by Desnoyers et. al. in \url{https://www.efficios.com/pub/rcu/urcu-suppl.pdf}, except we batch deallocation as in \url{http://software.imdea.org/~gotsman/papers/recycling-esop13.pdf}. We make a few simplifying assumptions:

- Counters used to store the grace period generation numbers cannot overflow.
- We will work with singly linked lists rather than doubly linked lists. Although this makes the update operation less general, it simplifies the proof without avoiding the main issues.
- There is a static limit $N$ on the number of readers.
- Reader threads never try to update entries in the list.
- We only prove safety properties; updaters can block indefinitely while waiting for readers to indicate they are in a quiescent state, and we do not prove the absence of memory leaks.
- There is a dedicated writer thread, and if a different thread becomes a writer, that ownership transfer is done through some external locking, which we don’t reason about here.

We assume there is some sequentially consistent implementation of a stack, with methods \texttt{newStack}, \texttt{push} and \texttt{pop}, which we use to batch the deallocation of nodes.

Besides the extra features we verify here, there are a few differences between the implementation here and the version presented in the body of the paper:

- We use \texttt{repeat .. until} instead of \texttt{while} or \texttt{for} loops.
- The formal language does not have tuples, so \texttt{rcuReadNext} passes its return values through parameter pointers rather than by returning a tuple, as is standard in C.
- Our \texttt{free} deallocated a word at a time, instead of the width of the original allocation at that pointer.
- We split \texttt{rcuSynchronize} into two pieces (\texttt{rcuSynchronize} and \texttt{rcuCollect}) because these are more natural points for giving specifications.
C.1 Code

rcuNew() \triangleq
let q = alloc(N + 4) /* q = generation counter, ptr to hd of list, number of readers, ptr to dealloc set, reader buffer */
[q + link]_{at} := 0;
let sc = alloc(1) /* Scratch space for counter – leaks memory */
[sc]_{na} := 0;
repeat /* somewhat awkward because there is no for loop primitive */
let i = [sc]_{na}
[q + rcounters + i]_{at} := 0
[sc]_{na} := i + 1;
i + 1 == N
end;
[q + wcounter]_{at} := 0;
[q + numreaders]_{at} := 0;
[q + del]_{na} := newStack();
q

registerReader(q) \triangleq
FAI(q + numreaders)

rcuQuiescentState(q, tid) \triangleq
let t = [q + wcounter]_{at}
[q + rcounters + tid]_{at} := t;
0

rcuSynchronize(q) \triangleq
let oldgc = [q + wcounter]_{at}
let newgc = oldgc + 1
[q + wcounter]_{at} := newgc;
let n = min(FAI(q + numreaders, 0), N) /* Fetch and increment by 0 */
rcuCollect(q, n, newgc)
0

rcuCollect(q, n, newgc) \triangleq
let sc = alloc(1)
[sc]_{na} := 0;
repeat let i = [sc]_{na}
repeat [q + rcounters + i]_{at} == newgc end
[sc]_{na} := i + 1;
i + 1 == n
end;
free(sc);
0

rcuNodeAppend(q, p, v) \triangleq
let x = alloc(2) /* node = value, child pointer */
[x + data]_{na} := v;
[x + link]_{at} := 0;
[p + link]_{at} := x;
x

rcuNodeUpdate(q, x, p, v) \triangleq
let c = [x + link]_{at}
let x' = alloc(2) /* node = value, child pointer */
[x' + data]_{na} := v;
[x' + link]_{at} := c;
[p + link]_{at} := x';
rcuDealloc(q, x);

rcuDealloc(q, x):
\[ x' \]

\[
\text{rcuNodeDel}(q, x, p) \triangleq \\
\text{let } c = [x + \text{link}]_{\text{at}} \\
[p + \text{link}]_{\text{at}} := c; \\
\text{rcuDealloc}(q, x); \\
\]

\[
\text{rcuReadStart}(q) \triangleq \\
[q + \text{link}]_{\text{acq}} \\
\text{rcuReadNext}(q, \text{nextptr}, \text{retptr}) \triangleq \\
/* \text{nextptr is a pointer to a pointer to a node (ie: node** nextptr)} */ \\
/* \text{returns 0 and stores value at that node in retptr if it exists} */ \\
/* \text{otherwise returns 1 if we're at the end of the list} */ \\
\text{let } p = [\text{nextptr}]_{\text{na}} \\
\text{if } p == 0 \text{ then 1} \\
\text{else} \\
\text{let } v = [p + \text{data}]_{\text{na}} \\
\text{let } p' = [p + \text{link}]_{\text{at}} \\
[\text{retptr}]_{\text{na}} := v; \\
[\text{nextptr}]_{\text{na}} := p' \\
0 \\
\]

\[
\text{rcuDealloc}(q, x) \\
\text{push}(q + \text{del}, x) \\
\text{let } c = \text{choose}(1, 2) \\
\text{if } c == 1 \text{ then 0} \\
\text{else} \\
\text{rcuSynchronize}(q) \\
\text{repeat} \\
\text{let } p = \text{pop}(q + \text{del}) \\
\text{if } p == 0 \text{ then 1} \\
\text{else} \\
\text{free}(p + \text{data}); \\
\text{free}(p + \text{link}); \\
0 \\
\end{verbatim}

C.2 Specification

We try to give a simple specification for the external methods in figure 8. We assume some fixed pure predicate \( P(x) \) that we require to hold of values in the list (e.g. they're all perfect squares). The specifications will require that when values are inserted, they satisfy this predicate, and that when we lookup a value, we get out something satisfying this predicate.

This specification involves several predicates that are abstract from the perspective of the client: \( \text{WriterSafe}(q, L) \), \( \text{ReaderSafe}(q, H, \text{tid}) \), \( \text{SafePtr}(H, p) \), and \( \text{ReaderQueue}(q) \). The first of these represents the permissions owned by the reader. It indicates that for the RCU structure with metadata at \( q \), the spine of the list consists of the pointers in the logical list \( L \). We generate this permission when we create a new RCU instance, and the writer’s methods simply modify the contents of \( L \) accordingly.

The reader’s analogous permission is \( \text{ReaderSafe}(q, H, \text{tid}) \). This \( H \) is not a list like in the writer’s predicate. Rather, \( H \) is some abstract type, and all clients can do is reason about equality of members of this type. This \( H \) corresponds to some sort of bound on the version of the list that the reader can see. \( \text{SafePtr}(q, H, p) \) means that \( p \) is a pointer that was valid with respect to this “bound” (or it is null). The reader can always get an initial \( \text{SafePtr} \) by calling \( \text{rcuReadStart} \), which simply reads \( q + \text{link} \). As the reader inspects the list using \( \text{rcuReadNext} \) it must feed in a valid \( \text{SafePtr} \), and it gets another one. However, when the
reader calls rcuQuiescentState it gets back ReaderSafe\((q, H, tid)\) for some fresh \(H\), thus making any previous SafePtr facts unusable. This forces the reader to start again at the head of the list through \(q + \text{link}\), and is what guarantees that the writer can safely de allocate any inaccessible nodes, since the reader is unable to try to use any old pointer to the dead node.

Finally, ReaderQueue\((q)\) is a duplicable predicate that indicates that \(q + \text{numreaders}\) is initialized and readers can try to register by calling registerReader\((q)\).

\section{C.3 Proof setup}

We will use a particular monoid to track the history of the list as ghost state. This monoid will allow a single writer to update the history of the list by modifying a non-duplicable “master” view of it. The monoid will also contain duplicable elements called “snapshots”, which are partial, possibly stale, histories of the object. Readers will use knowledge about these snapshots to reason about how the state of the list can evolve over time.

First, we have to define the traces that will represent the history of the list, and impose a partial order on them. Traces will be lists of abstract actions, where actions are of the form: upd\((i, i')\) or del\((i)\) where \(i \in \mathbb{N}\) and \(i' \in \mathbb{N} \cup \{\text{null}\}\). The former represents updating the link pointer for node \(i\) to point to node \(i'\) (if \(i' \in \mathbb{N}\)) or null (if \(i' = \text{null}\)). The latter indicates deallocating a node. Given a trace \(H\) and an abstract location \(i\), we can consider the subtrace of \(H\) containing only actions of the form

\begin{align*}
\{ \text{true} \} \\
\text{rcuNew}() \\
\{ q. H. WriterSafe(q, q + \text{link} \cdot \text{nil}) \ast \Box(ReaderQueue(q)) \}
\end{align*}

\begin{align*}
\{ \text{ReaderQueue}(q) \} \\
\text{registerReader}(q) \\
\{ x. \exists H. (x < N \ast \text{ReaderSafe}(q, H, x)) \lor (x \geq N) \}
\end{align*}

\begin{align*}
\{ \text{ReaderSafe}(q, -, tid) \} \\
\text{rcuQuiescentState}(q, tid) \\
\{ \exists H'. \text{ReaderSafe}(q, H', tid) \}
\end{align*}

\begin{align*}
\{ \text{ReaderSafe}(q, H, tid) \} \\
\text{rcuReadStart}(q) \\
\{ p. \text{ReaderSafe}(q, H, tid) \ast \Box\text{SafePtr}(q, H, p) \}
\end{align*}

\begin{align*}
\{ \text{WriterSafe}(q, L \cdot p \cdot L') \ast (L' \neq \text{nil}) \} \\
[p + \text{link}]_{\text{acq}} \\
\{ p'. \text{WriterSafe}(q, L \cdot p \cdot L') \ast \exists L'', L = p' \cdot L'' \}
\end{align*}

\begin{align*}
\{ \text{ReaderSafe}(q, H, tid) \ast \text{nextptr} \leftarrow p \ast \text{SafePtr}(q, H, p) \ast \text{retptr} \leftarrow \} \\
\text{rcuReadNext}(q, \text{nextptr}, \text{retptr}) \\
\{ x. \exists v, p'. \text{ReaderSafe}(q, H, tid) \ast \text{nextptr} \leftarrow p' \ast \Box(\text{SafePtr}(q, H, p')) \ast \text{retptr} \leftarrow v \ast ((x = 0 \land P(v)) \lor x = 1) \}
\end{align*}

\begin{align*}
\{ \text{WriterSafe}(q, L \cdot p) \ast \Box P(v) \} \\
\text{rcuNodeAppend}(q, p, v) \\
\{ x. \text{WriterSafe}(q, L \cdot p \cdot x) \}
\end{align*}

\begin{align*}
\{ \text{WriterSafe}(q, L \cdot p \cdot x \cdot L') \ast \Box P(v) \} \\
\text{rcuNodeUpdate}(q, x, p) \\
\{ x'. \text{WriterSafe}(q, L \cdot p \cdot x' \cdot L') \}
\end{align*}

\begin{align*}
\{ \text{WriterSafe}(q, L \cdot p \cdot x \cdot L') \} \\
\text{rcuNodeDel}(q, x, p) \\
\{ \text{WriterSafe}(q, L \cdot p \cdot L') \}
\end{align*}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{client_api.png}
\caption{Full specifications for client API}
\end{figure}

\section{2015/2/3}
We have the properties that \( \text{write the singleton permission} \) 
\[ \{ \text{where we order} \] 
\[ \text{the writer will use to retrieve these access tokens when it needs to deallocate a node, and} \] 
\[ \Gamma \] 
\[ \text{powerset of} \] 
\[ \text{containing all of the relevant monoid instances.} \] 

Given a trace \( H \), we can define a partial function \( H^* \) with a finite domain from \( \mathbb{N} \) to \( (\text{List } \mathbb{N} \cup \{\text{null}\}) \cup \{\top\} \) as follows:

\[
H^*(i) = \begin{cases} 
\top & \text{if } \text{del}(i) \in H \\
i_n \ldots i_1 \text{nul} & \text{if } H|_i = \text{upd}(i_1) \cdots \text{upd}(i_n) \quad (\text{so } H_i \text{ must be non-empty})
\end{cases}
\]

Note that the order of \( H^*(i) \) is opposite the order that the updates occurred, so that the head of the list is the most recent thing to which \( i \) points. We will often just drop the \( \ast \) and simply write \( H(i) \).

We order traces by saying that \( H_1 \leq H_2 \) if \( H_2 \) is a prefix of \( H_2 \). This is a preorder with the property that if \( H_1 \leq H_3 \) and \( H_2 \leq H_3 \) then \( H_1 \leq H_2 \) or \( H_2 \leq H_1 \). Note that if \( H_1 \leq H_2 \) then \( \text{dom}(H_1) \subseteq \text{dom}(H_2) \) and \( \forall i \in \text{dom}(H_1^*), H_1^*(i) \leq H_2^*(i) \) where we order \( \text{List } (\mathbb{N} \cup \{\text{null}\}) \cup \{\top\} \) by:

\[
L \leq \top \quad L_1 \leq L_2 \iff \exists L'. L_2 = L' \cdot L_1
\]

We say that a trace is in a valid configuration if:

- For all \( i, j, L \), \( H(i) = \top \land H(j) = i \cdot L \)
- For all \( i, j \), if \( \text{del}(i) \in H \), it occurs at most once, and is the last action of \( H \).
- For all \( i, i' \), if \( i' \in H(i) \) and \( i' \neq \text{null} \) then \( i' \in \text{dom}(H) \).

Together, these mean that no node currently points to a dead node or an uninitialized node, and we haven’t tried to update the pointer of a dead node or delete it twice. We say that \( H \) is well-formed if \( H \) and all of its prefixes are in valid configurations. From here onward, we restrict our attention to well-formed traces.

We define \( \text{dead}(H) \) and \( \text{live}(H) \) as the finite sets:

\[
\text{dead}(H) \triangleq \{i \mid H(i) = \top\} \\
\text{live}(H) \triangleq \text{dom}(H) \setminus \text{dead}(H)
\]

We have the properties that \( \text{live}(H) \cap \text{dead}(H) = \emptyset \) and if \( H \leq H' \), then \( \text{dead}(H) \subseteq \text{dead}(H') \). Hence if \( H \leq H' \), then \( \text{dead}(H) \cap \text{live}(H') = \emptyset \).

Notice that the restrictions we’ve placed on traces do not rule out cycles. In particular, a single node may be pointed to by two others, or itself. We have a separate predicate that captures this:

\[
\text{SinglePtr}(H) \triangleq (\forall i, i'. \text{hd}(H(i)) = \text{hd}(H(i')) \Rightarrow i = i') \land (\forall i. \text{hd}(H(i)) \neq i) \ast (\forall i. \text{hd}(H(i)) \neq \text{base}(H))
\]

The final conjunct here states that no node loops back to the metadata for the RCU instance, since \( \text{base}(H) \) corresponds to \( q + \text{link} \). This definition still allows abstract cycles of the form \( i \rightarrow j \rightarrow \cdots \rightarrow k \rightarrow i \), but these cycles must not be connected to the real list whose head is pointed to by \( q + \text{link} \), since we know that \( i \) cannot be \( \text{base}(H) \) and no node other than \( k \) can point to \( i \).

We can now define a PCM \( \mu \) which has elements of the form \( \text{Master}(H) \) and \( \text{Snapshot}(H) \) for each \( H \in \text{Traces} \). Then \( \mu \) is defined by:

\[
\text{Snapshot}(H) \ast \mu \text{Master}(H') \triangleq \text{Master}(H') \iff H \leq H' \\
\text{Master}(H) \ast \mu \text{Snapshot}(H') \triangleq \text{Master}(H) \iff H' \leq H \\
\text{Snapshot}(H) \ast \mu \text{Snapshot}(H') \triangleq \text{Snapshot}(\text{max}(H, H')) \text{ if this exists}
\]

We will need to split permissions for the data and \( \text{link} \) fields of nodes into partial pieces for readers (and one for the writer to retain). Our permissions will be subsets of \( \{0, \ldots, N\} \) where \( \top \) is the full set, and composition is defined as \( \ast \). We will often write the singleton permission \( \{n\} \) as \( n \).

Since we will work with a number of other ghost monoids, to simplify things we will use \( \Gamma \) as a metavariable for a record containing all of the relevant monoid instances. \( \Gamma \) history will refer to the instance of the above monoid \( \mu \). We also use the powerset of \( \{0, \ldots, N\} \times \mathbb{N} \) monoid with disjoint union as composition. We will work with three instances of this monoid: \( \Gamma_{\text{rxtok}} \), which represents the tokens used by threads to access elements of the linked list, \( \Gamma_{\text{wxtok}} \), which is for the tokens that the writer will use to retrieve these access tokens when it needs to deallocate a node, and \( \Gamma_{\text{ctok}} \) which are tokens used by the threads to update their version counter. The \( \{0, \ldots, N - 1\} \times \mathbb{N} \) of the access tokens and counter tokens are used by readers,
while \( \{N\} \times \mathbb{N} \) is reserved for the writer. Lastly, \( \Gamma \cdot q \) will correspond to the physical location that the RCU meta-data is located at.

We have a family of exchanges, \( \text{PermX}(\Gamma, l, tid, i) \) of the form:

\[
\text{PermX}(\Gamma, l, tid, i) \triangleq [\Gamma \cdot \text{rxtok} : \{(tid, i)\}] \rightsquigarrow (\exists v. l + \text{data} \xrightarrow{tid} v \cdot \text{P}(v)) \ast (l + \text{link} \xrightarrow{tid} i \ast \text{LLP}(\Gamma, i))
\]

These say that one can exchange an access token for fractional permission to read the \( l + \text{data} \) and \( l + \text{link} \) fields, and that the value stored in the former satisfies the predicate \( \text{P} \) that must hold of elements in the list.

We have an analogous exchange \( \text{ModX}(\Gamma, tid, i) \) for transferring \( \text{rxtok} \) and \( \text{wxtok} \):

\[
\text{ModX}(\Gamma, tid, i) \triangleq [\Gamma \cdot \text{rxtok} : \{(tid, i)\}] \rightsquigarrow [\Gamma \cdot \text{wxtok} : \{(tid, i)\}]
\]

When no possibility of ambiguity exists, we will sometimes inline these exchange definitions. When inlining them in this manner, we use an alternate notation, writing \( [P \lor Q] \) for knowledge that some exchange \( \sigma : P \rightsquigarrow Q \) exists.

First, we have a protocol \( \text{LLP}(\Gamma, i) \) where \( i \in \mathbb{N} \) for the \( \text{link} \) fields of the nodes. The states of this protocol are non-empty elements of \( \text{List} (\mathbb{N} \cup \{\text{null}\}) \), ordered as above. The protocol interpretation is:

\[
\text{LLP}(\Gamma, i)(\text{null} \cdot L, x) \triangleq x = 0 \ast \exists H. [\Gamma \cdot \text{history} : \text{Snapshot}(H)] \ast H(i) = \text{null} \cdot L
\]

\[
\text{LLP}(\Gamma, i)(i' \cdot L, x) \triangleq x \neq 0 \ast \exists H. [\Gamma \cdot \text{history} : \text{Snapshot}(H)] \ast H(i) = i' \cdot L \ast (\forall \text{tid} < N. \text{PermX}(\Gamma, x, tid, i'))
\]

We define a predicate \( \text{SnapshotValid}(\Gamma, H) \) which connects a snapshot to its implied interpretation about the state of \( \text{link} \) fields:

\[
\text{SnapshotValid}(\Gamma, H) \triangleq H \neq \text{null} \ast (\forall i \in \text{dom}(H), H(i) \neq \top \Rightarrow [i : H(i) \text{ LLP}(\Gamma, i)] \ast H(i) = \top \Rightarrow \exists s. i : s \text{ LLP}(\Gamma, i))
\]

\[
\ast [\Gamma \cdot \text{history} : \text{Snapshot}(H)] \ast \text{base}(H) \not\in \text{dead}(H)
\]

\[
\ast \exists \epsilon \in [0, 1]. \Gamma.q + \text{link} \xrightarrow{\epsilon} \text{base}(H)
\]

The second part of the statement about all \( i \in \text{dom}(H) \) seems strange it first. Its purpose is to merely state that if a node is marked as dead, then it was initialized at some point. The purpose of this will come up in \( \text{rebuildNodeUpdate} \) when the writer wants to add a fresh abstract location to the list. The last two parts state that \( \Gamma.q + \text{link} \) is the base of the trace and that it is not dead.

The protocol \( \text{WCP}(\Gamma) \) will be used for the writer’s generation counter. The states of the protocol will be elements of \( \text{Traces} \times \mathbb{N} \) ordered such that:

\[
(H, v) \leq (H', v') \iff (H = H' \land v \leq v') \lor (H \leq H' \land v < v')
\]

The interpretation is:

\[
\text{WCP}(\Gamma)((H, v), x) \triangleq x = v \ast [\Gamma \cdot \text{ctok} : \{(N, v)\}] \ast \text{SnapshotValid}(\Gamma, H)
\]

We have a corresponding protocol assertion for the generation counters for the readers, \( \text{RCP}(\Gamma, tid) \). The states are the same as for \( \text{WCP} \). The protocol interpretation is:

\[
\text{RCP}(\Gamma, tid)((H, v), x) \triangleq x = v \ast [\Gamma.q + \text{wc} : (H, v) \text{ WCP}(\Gamma)] \ast [\Gamma \cdot \text{ctok} : \{(tid, v)\}]
\]

\[
\ast (\forall i \in \text{dead}(H). [\Gamma \cdot \text{rxtok} : \{(tid, i)\}] \lor [\Gamma \cdot \text{wxtok} : \{(tid, i)\}])
\]

The idea is that a reader maintains the invariant that they have knowledge of some \( H \) such that \( \text{SnapshotValid}(\Gamma, H) \), and \( \forall i \in \mathbb{N} \) either the reader owns \([\Gamma \cdot \text{rxtok} : \{(tid, i)\}] \) or \( i \in \text{dead}(H) \). Now, consider what happens when a reader reads some link pointer at \( l + \text{link} \) corresponding to some abstract location \( i \in \text{live}(H) \) and learns that \( i \) is in state \( i' \cdot L \), where \( H(i) \leq i' \cdot L \) and there is some snapshot of an \( H' \) for which \( H'(i) = i' \cdot L \). Then we know that \( i' \not\in \text{dead}(H) \). To see this, note that either \( H \leq H' \) or \( H' \leq H \) since we must have that \( \text{Snapshot}(H) \cdot \text{Snapshot}(H') \) is well defined. Consider the two cases:

- \( H \leq H' \): Then \( i' \in \text{live}(H') \) (or else \( H' \) would not be well-formed), so \( i' \not\in \text{dead}(H) \), by the properties described above about the ordering of traces and deadsets.
• $H' \leq H$: Then we must have that $H(i) = i' \cdot L$ as well. Since if $H(i) \neq i' \cdot L$, we would have $H(i) < H'(i)$, which is impossible if $H' \leq H$. Hence, $i' \in \text{live}(H)$, so $i' \not\in \text{dead}(H)$.

so since $i \not\in \text{dead}(H)$, the reader must have $[\Gamma, \text{rxtok} : \{(\text{tid}, i)\}]$ as desired.

That is, if a thread knows a particular snapshot is valid, and has all of the rxtok for some tid, then it has “permission” to access the list. We define abstract predicates that captures this and other things the reader has to maintain:

$$\text{AccessList}(\Gamma, H, tid) \triangleq \text{SnapshotValid}(\Gamma, H) \land \{[\Gamma, \text{rxtok} : \{(\text{tid})\} \times (\mathbb{N} \setminus \text{dead}(H))] \}

\text{RCounterValid}(\Gamma, H, tid) \triangleq \exists v_1, v_2, H'. (H', v_1) \leq (H, v_2) \land [\Gamma, q + \text{wcounter} : (H, v_2)] \text{WCP}(\Gamma)

\begin{align*}
\text{ReaderSafe}(q, H, tid) & \triangleq \exists \Gamma. (q = q \land \text{AccessList}(\Gamma, H, tid) \land \text{RCounterValid}(\Gamma, H, tid)) \\
\text{SafePtr}(q, H, p) & \triangleq \exists \Gamma. (p = 0 \lor (p \\neq 0 \land \exists i'. i' \not\in \text{dead}(H) \land \text{PermX}(\Gamma, p, tid, i')))
\end{align*}

We have a protocol $\text{NRP}(\Gamma)$ on $\text{q} + \text{numreaders}$. The states of the protocol are the same as those of $\text{WCP}$ and $\text{RCP}$, but this time ordered pointwise. Initially, this protocol stores all of the ReaderSafe permissions for each possible reader thread. As readers register, they take out the permissions corresponding to the reader ID they're assigned. Similarly, when the writer wants to deallocate a node, it takes out the permissions from that node for the threads that haven’t yet registered.

$$\text{NRP}(\Gamma)((H, v), x) \triangleq x = v \land \text{nul}(\text{ReaderSafe}(\Gamma, q, H, tid))$$

The analogue of ReaderSafe for writers, WriterSafe($q, L$) is more complicated. Note that although the specification uses this predicate, internally we will work with WriterSafe’($q, L$). WriterSafe’($q, p_1 \cdot p_2 \cdot \cdots \cdot \text{nil}$) $\triangleq$ WriterSafe’($q, p_1 + \text{link} \cdot p_2 + \text{link} \cdot \cdots \cdot \text{nil}$). That is, we shift the pointers in the list to their link fields. The shifted form states that the most recent update done by the reader put the list in a state where the spine of the list contains the pointers in the abstract list $L$. Underneath, it states that there are three traces, $H_1, H_2,$ and $H_3$ such that the last time the writer’s counter was modified, the list trace was $H_1$, everything in the deallocation stack is in $\text{dead}(H_2) \setminus \text{dead}(H_1)$, the most up to date trace is $H_3$, and the writer’s counter and tokens adequately reflect these facts.

$$\text{RevokedUpTo}(\Gamma, H) \triangleq \exists v, v'. [\Gamma, q + \text{wcounter} : (H, v)] \text{WCP}(\Gamma) \land \{[\Gamma, \text{rxtok} : \{(\text{tid})\} \times (\mathbb{N} \setminus \text{dead}(H))] \}

\text{DeallocBetween}(\Gamma, H_1, H_2) \triangleq \exists L_d, L'_d. \text{Stack}(\Gamma, q + \text{del}, L_d) \land \text{NoDup}(L_d) \land \text{DeadFrom}(L_d, L'_d, H_1, H_2)

\text{CurrentState}(\Gamma, L, H) \triangleq \{[\Gamma, \text{history} : \text{Master}(\Gamma)] \land \text{SnapshotValid}(H) \land \text{TraceSpine}(\Gamma, H, L) \land \text{SinglePtr}(H)

\text{WriterSafe’}(q, L) \triangleq \exists \Gamma. (H_1, H_2, H_3. q = q \land H_1 \leq H_2 \leq H_3 \land \text{RevokedUpTo}(\Gamma, H_1))

\text{DeallocBetween}(\Gamma, H_1, H_2) \land \text{CurrentState}(\Gamma, L, H_3)

\text{NoDup}(L) \triangleq \forall i, i'. L[i] = L[i'] \Rightarrow i = i'

\text{TraceSpine}(\Gamma, H, L) \triangleq \exists L'. |L'| = |L| \land (\forall i < |L'|. L'[i] \in \text{live}(H) \land \text{hd}(H(L'[i]))) = L'[i + 1])

\text{DeadFrom}(\Gamma, L_d, L'_d, H_1, H_2) \triangleq |L_d| = |L'_d| \land (\forall j \geq |L_d|. L'_d[j] \in \text{dead}(H_2) \setminus \text{dead}(H_1) \land \forall n < N. \text{PermX}(\Gamma, L_d[j], n, L'_d[j]))
C.4 Hoare proofs

C.4.1 Verification of rcuNew

The specification for rcuNew says that at the end we get all of the permissions that the writer thread needs. From there, we could transfer these to some other writer, and transmit knowledge of the \( q + \text{numreaders} \) counter to readers through whatever means we like.

\[
\{\text{true}\}
\]

\[
\text{rcuNew}()
\]

\[
\{ q.\text{WriterSafe}(q, q + \text{link} \cdot \text{nil}) \ast \text{ReaderQueue}(q) \}
\]

For concision, we use the following abstract predicate in this proof:

\[
\text{RCountersInitUpTo}(\Gamma, \varphi, j) \triangleq \{ \Gamma.\text{history} : \text{Master}([\varphi, \text{null}]) \} \ast \{ \Gamma.\text{xtok} : \{0, \ldots, N\} \times N \}
\]

\[
\ast \{ \Gamma.\text{xtok} : \{0, \ldots, j\} \times N \} \ast \{ \Gamma.\text{ctok} : \{0, \ldots, j - 1\} \times (N \setminus \{0\}) \}
\]

\[
\ast \left( \bigstar_{tid < j} (\Gamma.q + \text{rcounters} + \text{tid} : (\text{upd}(\varphi, \text{null}), 0) \text{ RCP}(\Gamma)) \right)
\]

\[
\ast \left( \bigstar_{j \leq tid < N} \text{uninit}(\Gamma.q + \text{rcounters} + \text{tid}) \right)
\]

\[
\{\text{true}\}
\]

\[
\text{let } q = \text{alloc}(N + 4) \quad /\* q = \text{generation counter, ptr to head of list, number of readers, ptr to dealloc set, reader buffer } */
\]

\[
\square(q \neq 0) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{link}) \ast \text{uninit}(q + \text{numreaders}) \ast \text{uninit}(q + \text{del})
\]

\[
\ast \left( \bigstar_{tid < N} \text{uninit}(q + \text{rcounters} + \text{tid}) \right)
\]

\[
\exists \varphi. \text{uninit}(\varphi) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{link}) \ast \text{uninit}(q + \text{numreaders}) \ast \text{uninit}(q + \text{del})
\]

\[
\ast \left( \bigstar_{tid < N} \text{uninit}(q + \text{rcounters} + \text{tid}) \right)
\]

\[
\text{uninit}(\varphi) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{link}) \ast \text{uninit}(q + \text{numreaders}) \ast \text{uninit}(q + \text{del})
\]

\[
\ast \left( \bigstar_{tid < N} \text{uninit}(q + \text{rcounters} + \text{tid}) \right)
\]

\[
\exists \Gamma. \text{RCountersInitUpTo}(\Gamma, \varphi, 0) \ast \Gamma.q = q \ast \text{uninit}(\varphi) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{link})
\]

\[
\ast \text{uninit}(q + \text{numreaders}) \ast \text{uninit}(q + \text{del})
\]

\[
\text{RCountersInitUpTo}(\Gamma, \varphi, 0) \ast \square(\Gamma.q = q) \ast \text{uninit}(\varphi) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{link})
\]

\[
\ast \text{uninit}(q + \text{numreaders}) \ast \text{uninit}(q + \text{del})
\]

\[
\{ q + \text{link}[a] := 0; \}
\]

\[
\text{RCountersInitUpTo}(\Gamma, \varphi, 0) \ast \square(\varphi : \text{null} \cdot \text{nil} \text{ LLP}(\Gamma, \varphi)) \ast \square(\exists \epsilon \in [0, 1], q + \text{link} \sim \varphi)
\]

\[
\ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{numreaders}) \ast \text{uninit}(q + \text{del})
\]

\[
\text{RCountersInitUpTo}(\Gamma, \varphi, 0) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{numreaders}) \ast \text{uninit}(q + \text{del})
\]

\[
\text{let } sc = \text{alloc}(1) \quad /\* \text{Scratch space for counter } – \text{leaks memory } */
\]

\[
\text{RCountersInitUpTo}(\Gamma, \varphi, 0) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{numreaders}) \ast \text{uninit}(q + \text{del})
\]

\[
\ast \text{uninit}(sc)
\]

\[
[sc]_{\text{a}} := 0;
\]
\[
\begin{align*}
\text{RCountersInitUpTo}(\Gamma, \varphi, 0) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{numreaders}) \\
\ast \text{uninit}(q + \text{del}) \ast \text{sc} \mapsto 0
\end{align*}
\]

\[
\exists c. \text{RCountersInitUpTo}(\Gamma, \varphi, c) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{numreaders}) \\
\ast \text{uninit}(q + \text{del}) \ast \text{sc} \mapsto c
\]

repeat

let \(i = [\text{sc}]_{\text{na}}\)

\[
\begin{align*}
\text{RCountersInitUpTo}(\Gamma, \varphi, i) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{numreaders}) \\
\ast \text{uninit}(q + \text{del}) \ast \text{sc} \mapsto i
\end{align*}
\]

\([q + \text{rcounters} + i]_{\text{at}} := 0\)

\[
\begin{align*}
\text{RCountersInitUpTo}(\Gamma, \varphi, i + 1) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{numreaders}) \\
\ast \text{uninit}(q + \text{del}) \ast \text{sc} \mapsto i
\end{align*}
\]

\([\text{sc}]_{\text{na}} := i + 1;\)

\[
\begin{align*}
\text{RCountersInitUpTo}(\Gamma, \varphi, i + 1) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{numreaders}) \\
\ast \text{uninit}(q + \text{del}) \ast \text{sc} \mapsto i + 1
\end{align*}
\]

\(i + 1 \equiv N\)

end:

\[
\begin{align*}
\text{RCountersInitUpTo}(\Gamma, \varphi, N) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{numreaders}) \\
\ast \text{uninit}(q + \text{del}) \ast \text{uninit}(q + \text{wcounter}) \ast \text{uninit}(q + \text{numreaders}) \ast \text{uninit}(q + \text{del})
\end{align*}
\]

\[
\begin{align*}
\Gamma.\text{history} : \text{Master}(\text{upd}(\varphi, \text{null})) \ast \Gamma.\text{rxtok} : \{0, \ldots, N\} \times \mathbb{N} \ast \Gamma.\text{wxtok} : \{0, \ldots, N\} \times \mathbb{N}
\end{align*}
\]

\[
\begin{align*}
\Gamma.\text{ctok} : \{0, \ldots, N \} \times (\mathbb{N} \setminus \{0\}) \ast \Gamma.\text{ctok} : \{0, \ldots, N\} \times \mathbb{N}
\end{align*}
\]

\[
\begin{align*}
\top \left( \ast_{\text{tid} \in \mathbb{N}} \Gamma.q + \text{rcounters} + \text{tid} : (\text{upd}(\varphi, \text{null}), 0) \ast \text{RCP}(\Gamma) \right)
\end{align*}
\]

\([q + \text{wcounter}]_{\text{at}} := 0;\)

\[
\begin{align*}
\top \left( q + \text{wcounter} : (\text{upd}(\varphi, \text{null}), 0) \ast \text{WCP}(\Gamma) \right) \ast \text{uninit}(q + \text{numreaders}) \ast \text{uninit}(q + \text{del})
\end{align*}
\]

\[
\begin{align*}
\Gamma.\text{history} : \text{Master}(\text{upd}(\varphi, \text{null})) \ast \Gamma.\text{rxtok} : \{0, \ldots, N\} \times \mathbb{N} \ast \Gamma.\text{wxtok} : \{0, \ldots, N\} \times \mathbb{N}
\end{align*}
\]

\[
\begin{align*}
\Gamma.\text{ctok} : \{0, \ldots, N\} \times (\mathbb{N} \setminus \{0\}) \ast \Gamma.\text{ctok} : \{0, \ldots, N\} \times (\mathbb{N} \setminus \{0\})
\end{align*}
\]

\([q + \text{numreaders}]_{\text{at}} := 0;\)

\[
\begin{align*}
\top \left( q + \text{numreaders} : (\text{upd}(\varphi, \text{null}), 0) \ast \text{NRP}(\Gamma) \right) \ast \text{uninit}(q + \text{del})
\end{align*}
\]

\[
\begin{align*}
\Gamma.\text{history} : \text{Master}(\text{upd}(\varphi, \text{null})) \ast \Gamma.\text{rxtok} : \{N\} \times \mathbb{N} \ast \Gamma.\text{wxtok} : \{0, \ldots, N\} \times \mathbb{N}
\end{align*}
\]

\[
\begin{align*}
\Gamma.\text{ctok} : \{N\} \times (\mathbb{N} \setminus \{0\}) \ast \Gamma.\text{ctok} : \{N\} \times (\mathbb{N} \setminus \{0\})
\end{align*}
\]

\([q + \text{del}]_{\text{na}} := \text{newStack}();\)

\[
\begin{align*}
\text{Stack}(q + \text{del}, \text{nil}) \ast \text{NoDup}(\text{nil}) \ast \text{DeadFrom}(\text{nil}, \text{nil}, \text{upd}(\varphi, \text{null}), \text{upd}(\varphi, \text{null}))
\end{align*}
\]

\[
\begin{align*}
\Gamma.\text{history} : \text{Master}(\text{upd}(\varphi, \text{null})) \ast \Gamma.\text{rxtok} : \{N\} \times \mathbb{N} \ast \Gamma.\text{wxtok} : \{0, \ldots, N\} \times \mathbb{N}
\end{align*}
\]

\[
\begin{align*}
\Gamma.\text{ctok} : \{N\} \times (\mathbb{N} \setminus \{0\}) \ast \text{TraceSpine}(\text{upd}(\varphi, \text{null}), q + \text{link} \cdot \text{nil})
\end{align*}
\]

\[
\text{WriterSafe}'(q, q + \text{link} \cdot \text{nil}) \ast \text{ReaderQueue}(q)
\]

\(q\)

\[
\begin{align*}
\{ q.\text{WriterSafe}'(q, q + \text{link} \cdot \text{nil}) \ast \text{ReaderQueue}(q) \}
\end{align*}
\]
\subsection{Verification of registerReader}

\begin{equation*}
\{ \text{ReaderQueue}(q) \} \\
\text{registerReader}(q) \\
\{ x. \exists H. (x < N \ast \text{ReaderSafe}(q, H, x)) \lor (x \geq N) \}
\end{equation*}

This function consists of a single FAI. The key is that when we perform the FAI, if the current value \( v \) is less than \( N \), then there is a ReaderSafe(\( \Gamma \), \( H \), \( v \)) stored in the protocol, which the reader is allowed to take out as part of the FAI.

\subsection{Verification of rcuReadNext}

\begin{equation*}
\{ \exists p. \text{ReaderSafe}(q, H, tid) \ast \text{nextptr} \rightarrow p \ast \text{SafePtr}(H, p) \ast \text{retptr} \leftarrow - \}
\end{equation*}

\begin{align*}
\text{rcuReadNext}(q, \text{nextptr}, \text{retptr}) \\
\{ x. \exists v. \text{ReaderSafe}(q, H, tid) \ast \text{nextptr} \rightarrow p \ast \text{SafePtr}(H, p) \ast \text{retptr} \leftarrow v \ast ((x = 0 \land P(v)) \lor x = 1) \}
\end{align*}

First, as a technicality, note that:

\[ \text{SnapshotValid}(\Gamma, H) \ast \text{SnapshotValid}(\Gamma', H) \Rightarrow \Gamma = \Gamma' \]

To see this, note the former implies that \[ \text{base}(H) : H(\text{base}(H)) \ast \text{LLP}(\Gamma, H(\text{base}(H))) \] while the latter implies \[ \text{base}(H) : H(\text{base}(H)) \ast \text{LLP}(\Gamma', H(\text{base}(H))) \] An atomic location can be bound to only one protocol, so these two protocols must be the same, so \( \Gamma = \Gamma' \).

\begin{align*}
\{ \exists p. \text{ReaderSafe}(q, H, tid) \ast \text{nextptr} \rightarrow p \ast \text{SafePtr}(H, p) \ast \text{retptr} \leftarrow - \}
\end{align*}

\begin{align*}
\text{let } p = [\text{nextptr}]_{na} \\
\{ \text{ReaderSafe}(q, H, tid) \ast \text{nextptr} \rightarrow p \ast \text{SafePtr}(H, p) \ast \text{retptr} \leftarrow - \}
\end{align*}

\begin{align*}
\exists ! \Gamma. q = q \ast \text{AccessList}(\Gamma, H, tid) \ast \text{RCounterValid}(\Gamma, H, tid) \ast \text{nextptr} \rightarrow p \ast \text{SafePtr}(H, p) \ast \text{retptr} \leftarrow - \\
\Box(\text{SnapshotValid}(\Gamma, H)) \ast \text{Γ'.rxtok : \{tid\} × (N \setminus \text{dead}(H))} \ast \Box\left( (\exists \in [0, 1]. \Gamma.q + \text{link} \rightarrow \text{base}(H)) \right)
\end{align*}

\begin{align*}
\{ \Gamma'.rxtok : \{\text{tid}\} × (N \setminus \text{dead}(H)) \} \ast \text{RCounterValid}(\Gamma, H, tid) \ast \text{nextptr} \rightarrow p \ast \text{SafePtr}(H, p) \ast \text{retptr} \leftarrow - \\
\ast \Box(\Gamma'. \text{SnapshotValid}(\Gamma', H)) \ast ((p = 0) \lor (p \neq 0 \ast \exists i. i \notin \text{dead}(H) \ast \text{PermX}(\Gamma', p, tid, i)))
\end{align*}

\begin{align*}
\{ \Gamma'.rxtok : \{\text{tid}\} × (N \setminus \text{dead}(H)) \} \ast \text{RCounterValid}(\Gamma, H, tid) \ast \text{nextptr} \rightarrow p \ast \text{retptr} \leftarrow - \\
\ast \Box(\Gamma'. \text{SnapshotValid}(\Gamma', H)) \ast ((p = 0) \lor (p \neq 0 \ast \exists i. i \notin \text{dead}(H) \ast \text{PermX}(\Gamma', p, tid, i))) \ast \Box(\Gamma = \Gamma')
\end{align*}

\begin{align*}
\{ \Gamma'.rxtok : \{\text{tid}\} × (N \setminus \text{dead}(H)) \} \ast \text{RCounterValid}(\Gamma, H, tid) \ast \text{nextptr} \rightarrow p \ast \text{retptr} \leftarrow - \\
\ast \Box(\Gamma'. \text{SnapshotValid}(\Gamma', H)) \ast ((p = 0) \lor (p \neq 0 \ast \exists i. i \notin \text{dead}(H) \ast \text{PermX}(\Gamma, p, tid, i)))
\end{align*}

if \( p = 0 \) then 1

else

\begin{align*}
\{ \Gamma'.rxtok : \{\text{tid}\} × (N \setminus \text{dead}(H)) \} \ast \text{RCounterValid}(\Gamma, H, tid) \ast \text{nextptr} \rightarrow p \ast \text{retptr} \leftarrow - \\
\ast \Box(\Gamma'. \text{SnapshotValid}(\Gamma', H)) \ast ((p = 0) \lor (p \neq 0 \ast \exists i. i \notin \text{dead}(H) \ast \text{PermX}(\Gamma, p, tid, i)))
\end{align*}
\[
\begin{align*}
\Gamma.\text{rxtok} &: \{(\text{tid}) \times (\mathbb{N} \setminus \text{dead}(H))\} \Rightarrow \text{RCounterValid}(\Gamma, H, \text{tid}) \quad \text{nextptr} \mapsto p \ast \text{retptr} \mapsto - \\
& \quad \ast \Box (i \not\in \text{dead}(H) \ast \text{PermX}(\Gamma, p, \text{tid}, i)) \\
\Gamma.\text{rxtok} &: \{(\text{tid}) \times ((\mathbb{N} \setminus \text{dead}(H)) \setminus \{i\})\} \Rightarrow \text{RCounterValid}(\Gamma, H, \text{tid}) \\
& \quad \ast \text{nextptr} \mapsto p \ast \text{retptr} \mapsto - \\
\Gamma.\text{rxtok} &: \{(\text{tid}) \times ((\mathbb{N} \setminus \text{dead}(H)) \setminus \{i\})\} \Rightarrow \text{RCounterValid}(\Gamma, H, \text{tid}) \\
& \quad \left(\exists v. p \ast \text{data} \mapsto v \ast P(v) \ast (p + \text{link} \mapsto \text{id} \ast i \ast \text{null LLP}(\Gamma, i))\right) \\
& \quad \ast \text{nextptr} \mapsto p \ast \text{retptr} \mapsto - \\
\end{align*}
\]
since this is what allows us to first get a SafePtr with which we can call rcuReadNext.

\[
\{ \text{ReaderSafe}(q, H, tid) \}
\]

\[
\exists H. \text{ReaderSafe}(q, H, tid)
\]

\[
\text{ReaderSafe}(q, H, tid)
\]

\[
\exists H. \text{ReaderSafe}(q, H, tid)
\]

\[
\{ \exists H. \text{ReaderSafe}(q, H, tid) \}
\]

\[
\text{rcuQuiescentState}(q, tid)
\]

\[
\exists H'. \text{ReaderSafe}(q, H', tid)
\]

\[
\{ \exists H. \text{ReaderSafe}(q, H, tid) \}
\]

\[
\text{ReaderSafe}(q, H, tid)
\]

\[
\exists \Gamma. q = q \ast \text{AccessList}(\Gamma, H, tid) \ast \text{RCounterValid}(\Gamma, H, tid)
\]

\[
\{ \exists \Gamma. q = q \ast \text{AccessList}(\Gamma, H, tid) \ast \text{RCounterValid}(\Gamma, H, tid) \}
\]

\[
\text{SnapshotValid}(\Gamma, H) \ast \{ \text{Γ.rxtok : {tid} \ast \{N \setminus \text{dead}(H)\} \ast \text{RCounterValid}(\Gamma, H, tid) \}
\]

\[
\exists (\text{SnapshotValid}(\Gamma, H)) \ast \{ \text{Γ.rxtok : {tid} \ast \{N \setminus \text{dead}(H)\} \ast \text{RCounterValid}(\Gamma, H, tid) \}
\]

\[
\\square(\text{SnapshotValid}(\Gamma, H)) \ast \{ \text{Γ.rxtok : {tid} \ast \{N \setminus \text{dead}(H)\} \ast \text{RCounterValid}(\Gamma, H, tid) \}
\]

\[
\exists (\text{SnapshotValid}(\Gamma, H)) \ast \{ \text{Γ.rxtok : {tid} \ast \{N \setminus \text{dead}(H)\} \ast \text{RCounterValid}(\Gamma, H, tid) \}
\]

\[
\{ \text{Γ.rxtok : {tid} \ast \{N \setminus \text{dead}(H)\} \ast \text{RCounterValid}(\Gamma, H, tid) \}
\]

\[
\{ \exists H. \text{ReaderSafe}(q, H, tid) \}
\]

\[
\text{rcuQuiescentState}(q, tid)
\]

\[
\{ \exists H'. \text{ReaderSafe}(q, H', tid) \}
\]

\[
\text{ReaderSafe}(q, H, tid)
\]

\[
\exists \Gamma. q = q \ast \text{AccessList}(\Gamma, H, tid) \ast \text{RCounterValid}(\Gamma, H, tid)
\]

\[
\{ \exists \Gamma. q = q \ast \text{AccessList}(\Gamma, H, tid) \ast \text{RCounterValid}(\Gamma, H, tid) \}
\]

\[
\text{SnapshotValid}(\Gamma, H) \ast \{ \text{Γ.rxtok : {tid} \ast \{N \setminus \text{dead}(H)\} \ast \text{RCounterValid}(\Gamma, H, tid) \}
\]

\[
\exists (\text{SnapshotValid}(\Gamma, H)) \ast \{ \text{Γ.rxtok : {tid} \ast \{N \setminus \text{dead}(H)\} \ast \text{RCounterValid}(\Gamma, H, tid) \}
\]

\[
\\square(\text{SnapshotValid}(\Gamma, H)) \ast \{ \text{Γ.rxtok : {tid} \ast \{N \setminus \text{dead}(H)\} \ast \text{RCounterValid}(\Gamma, H, tid) \}
\]

\[
\exists (\text{SnapshotValid}(\Gamma, H)) \ast \{ \text{Γ.rxtok : {tid} \ast \{N \setminus \text{dead}(H)\} \ast \text{RCounterValid}(\Gamma, H, tid) \}
\]

\[
\{ \text{Γ.rxtok : {tid} \ast \{N \setminus \text{dead}(H)\} \ast \text{RCounterValid}(\Gamma, H, tid) \}
\]

\[
\exists H. \text{ReaderSafe}(q, H, tid)
\]

Notice that we could give this spec to a function which did nothing. Because we don’t prove liveness properties, this spec doesn’t guarantee that the writer can make progress, even if the readers do call rcuQuiescentState.
let \( t = [q + \text{wcounter}]_{\text{at}} \)

\[
\begin{align*}
\Gamma. \text{rxtok} &: \langle \text{tid} \rangle \times (N \setminus \text{dead}(H)) \land \Gamma. \text{ctok} : \langle \text{tid} \rangle \times (N \setminus \{0, \ldots, v_i\}) \\
&* \exists H'', v_3. t = v_3 * (H, v_2) \leq (H'', v_3) * [q + \text{wcounter} : (H'', v_3)] \text{WCP}(\Gamma) * \text{SnapshotValid}(\Gamma, H'') \\
\Gamma. \text{rxtok} &: \langle \text{tid} \rangle \times (N \setminus \text{dead}(H')) \land \Gamma. \text{ctok} : \langle \text{tid} \rangle \times (N \setminus \{0, \ldots, v_i - 1\}) \\
&* \Box (t = v_3) * \Box ((H, v_2) \leq (H', v_3)) * \Box [q + \text{wcounter} : (H', v_3)] \text{WCP}(\Gamma) * \Box \text{SnapshotValid}(\Gamma, H'') \\
\Gamma. \text{rxtok} &: \langle \text{tid} \rangle \times (N \setminus \text{dead}(H'')) \land \Gamma. \text{ctok} : \langle \text{tid} \rangle \times (N \setminus \{0, \ldots, v_i\}) \\
&* \Box (\bigvee_{i \in \text{dead}(H'')} \Gamma. \text{rxtok} : \langle \text{tid}, i \rangle) \land \Box [q + \text{wcounters + tid} : (H'', v_3)] \text{RCP}(\Gamma, tid) \\
\{ \text{ReaderSafe}(q, H'', tid) \}
\end{align*}
\]

0

\[ \exists H. \text{ReaderSafe}(\Gamma, H, tid) \]

### C.4.6 Verification of \text{rcuCollect}

This routine collects tokens for dead nodes from currently registered readers up to reader id \( n \). It is called by \text{rcuSynchronize}, which collects the tokens of the unregistered readers via a read-modify-write on \( q + \text{numreaders} \). The specification and proof are parameterized by two traces, \( H_1 \) and \( H_2 \) (that is, we are proving here a family of triples). We exchange the \text{wxtok} for \text{rxtok} for each \( i \) which is dead in \( H_2 \) but not \( H_1 \).

\[
\begin{align*}
\Gamma. q = q + \text{wcounter} : (H_2, \text{newgc}) \text{WCP}(\Gamma) * H_1 \leq H_2 * \{ \Gamma. \text{history} : \text{Master}(H_2) \} \\
&* \Gamma. \text{wxtok} : \langle 0, \ldots, n - 1 \rangle \times (\text{dead}(H_2) \setminus \text{dead}(H_1)) * \forall tid < n. [q + \text{wcounters + tid} : - \text{RCP}(\Gamma, tid)] \\
\text{rcuCollect}(q, n, \text{newgc})
\end{align*}
\]

\[
\begin{align*}
\Box (\Gamma. q = q) &* \Box (q + \text{wcounter} : (H_2, \text{newgc}) \text{WCP}(\Gamma)) * \Box (H_1 \leq H_2) * \{ \Gamma. \text{history} : \text{Master}(H_2) \} \\
&* \Box \Gamma. \text{wxtok} : \langle 0, \ldots, n - 1 \rangle \times (\text{dead}(H_2) \setminus \text{dead}(H_1)) * \Box (\forall tid < n. [q + \text{wcounters + tid} : - \text{RCP}(\Gamma, tid)]
\end{align*}
\]

let \( sc = \text{alloc}(1) \)

\[ [sc]_{\text{na}} = 0; \]

\[
\begin{align*}
\{ \Gamma. \text{history} : \text{Master}(H_2) \} &* \Gamma. \text{wxtok} : \langle 0, \ldots, n - 1 \rangle \times (\text{dead}(H_2) \setminus \text{dead}(H_1)) * sc \leftarrow 0
\end{align*}
\]

\[ \exists k. [\Gamma. \text{history} : \text{Master}(H_2)] * sc \leftarrow k
\]

\[
\begin{align*}
&* \Gamma. \text{wxtok} : \langle k, \ldots, n - 1 \rangle \times (\text{dead}(H_2) \setminus \text{dead}(H_1)) * \Gamma. \text{wxtok} : \langle 0, \ldots, k - 1 \rangle \times (\text{dead}(H_2) \setminus \text{dead}(H_1))
\end{align*}
\]

repeat  // somewhat awkward because there is no for loop primitive */

let \( i = [sc]_{\text{na}} \)
\[\begin{align*}
&\{\Gamma.\text{history} : \text{Master}(H_2)\} \times sc \mapsto i \\
&\{\Gamma.\text{wxtok} : \{i, \ldots, n-1\} \times (\text{dead}(H_2) \setminus \text{dead}(H_1))\} \times \{\Gamma.\text{wxtok} : \{0, \ldots, i-1\} \times (\text{dead}(H_2) \setminus \text{dead}(H_1))\}\end{align*}\]

repeat \([q+rcounters+i]_a = newgc\) end

\[\begin{align*}
&\{\Gamma.\text{history} : \text{Master}(H_2)\} \times sc \mapsto i \\
&\{\Gamma.\text{wxtok} : \{i, \ldots, n-1\} \times (\text{dead}(H_2) \setminus \text{dead}(H_1))\} \times \{\Gamma.\text{wxtok} : \{0, \ldots, i-1\} \times (\text{dead}(H_2) \setminus \text{dead}(H_1))\}\end{align*}\]

\[\exists H', v', v' = newgc \times q + \text{counter} : (H', v') \ WCP(\Gamma) \]

\[\big(\forall i \in \text{dead}(H'). \Gamma.\text{rxtok} : \{\text{tid}, i\}\big) \lor \Gamma.\text{wxtok} : \{\text{tid}, i\}\]

\[\{\Gamma.\text{history} : \text{Master}(H_2)\} \times sc \mapsto i \\
\{\Gamma.\text{wxtok} : \{i, \ldots, n-1\} \times (\text{dead}(H_2) \setminus \text{dead}(H_1))\} \times \{\Gamma.\text{wxtok} : \{0, \ldots, i-1\} \times (\text{dead}(H_2) \setminus \text{dead}(H_1))\}\]

\[\begin{align*}
&\begin{cases}
\exists i. \{\Gamma.\text{history} : \text{Master}(H_2)\} \times i + 1 = n \times sc \mapsto i \\
\{\Gamma.\text{wxtok} : \{i, \ldots, n-1\} \times (\text{dead}(H_2) \setminus \text{dead}(H_1))\} \times \{\Gamma.\text{wxtok} : \{0, \ldots, i-1\} \times (\text{dead}(H_2) \setminus \text{dead}(H_1))\}\end{cases}
\end{align*}\]

\[\{\Gamma.\text{history} : \text{Master}(H_2)\} \times sc \mapsto n \times 1 \times \{\Gamma.\text{wxtok} : \{0, \ldots, n-1\} \times (\text{dead}(H_2) \setminus \text{dead}(H_1))\}\]

\[\{\text{free}(\text{sc})\}
\]

\[\{\text{Gamma.history} : \text{Master}(H_2)\} \times \{\text{Gamma.wxtok} : \{0, \ldots, n-1\} \times (\text{dead}(H_2) \setminus \text{dead}(H_1))\}\]

0

C.4.7 Verification of rcuSynchronize

The specification and proof are parameterized by two traces, \(H_1\) and \(H_2\) (that is, we are proving here a family of triples). rcuSynchronize exchanges all of the wxtok for rxtok for each \(i\) which is dead in \(H_2\) but not \(H_1\). In rcuNodeUpdate, this will be instantiated to some particular \(H_1\) and \(H_2\), and then the caller will use the rxtok to get access to the node it wants to deallocate.

\[\begin{align*}
&\exists \Gamma. q = q \times H_1 \leq H_2 \times \text{RevokedUpTo}(\Gamma, H_1) \times \text{SnapshotValid}(\Gamma, H_2) \times \{\Gamma.\text{history} : \text{Master}(H_2)\}\}
\text{rcuSynchronize}(q)
\end{align*}\]

\[\begin{align*}
&\exists \Gamma. q = q \times \text{RevokedUpTo}(\Gamma, H_2) \times \{\Gamma.\text{history} : \text{Master}(H_2)\} \times \{\Gamma.\text{rxtok} : \{0, \ldots, N\} \times (\text{dead}(H_2) \setminus \text{dead}(H_1))\}\}
\end{align*}\]
\[
\exists \Gamma. \Gamma.q = q \land \text{RevokedUpTo}(\Gamma, H_1) \land H_1 \leq H_2 \land \text{SnapshotValid}(\Gamma, H_2) \land \Gamma.\text{history} : \text{Master}(H_2)
\]

\[
\boxdot(\Gamma.q = q) \land \text{RevokedUpTo}(\Gamma, H_1) \land \boxdot(H_1 \leq H_2) \land \boxdot(\text{SnapshotValid}(\Gamma, H_2)) \land \Gamma.\text{history} : \text{Master}(H_2)
\]

\[
\exists v_1, v_2. \quad q + \text{wcounter} : (H_1, v_1) \land \text{WCP}(\Gamma) \land \Gamma.\text{ctok} : \{N\} \times \{N \setminus 0, \ldots, v_1\} \land \Gamma.wxtok : (0, \ldots, N - 1) \times (\text{dead}(H_2) \setminus \text{dead}(H_1))
\]

\[
\exists \Gamma. \Gamma.q = \Gamma.q \land \Gamma.wcounter : (H_1, v_1) \land \text{WCP}(\Gamma) \land \Gamma.\text{ctok} : \{N\} \times \{N \setminus 0, \ldots, v_1\} \land \Gamma.wxtok : (0, \ldots, N - 1) \times (\text{dead}(H_2) \setminus \text{dead}(H_1))
\]

\[
\Gamma.\text{ctok} : \{N\} \times \{N \setminus 0, \ldots, v_1\} \land \Gamma.\text{history} : \text{Master}(H_2) \land \Gamma.wxtok : (0, \ldots, N - 1) \times (\text{dead}(H_2) \setminus \text{dead}(H_1))
\]

\[
\text{AccessList}(\Gamma, H_1, N) \land \boxdot(\text{oldgc} = v_1)
\]

let oldgc = \[q + \text{wcounter}]_\text{at}
\[
\Gamma.\text{ctok} : \{N\} \times \{N \setminus 0, \ldots, v_1\} \land \Gamma.\text{history} : \text{Master}(H_2) \land \Gamma.wxtok : (0, \ldots, N - 1) \times (\text{dead}(H_2) \setminus \text{dead}(H_1))
\]

\[
\text{AccessList}(\Gamma, H_1, N) \land \boxdot(\text{newgc} = v_1 + 1)
\]

\[
[q + \text{wcounter}]_\text{at} := \text{newgc};
\]

\[
\Gamma.\text{ctok} : \{N\} \times \{N \setminus 0, \ldots, v_1 + 1\} \land \Gamma.\text{history} : \text{Master}(H_2) \land \Gamma.wxtok : (0, \ldots, N - 1) \times (\text{dead}(H_2) \setminus \text{dead}(H_1))
\]

\[
\text{AccessList}(\Gamma, H_1, N) \land \boxdot(q + \text{wcounter} : (H_2, v_1 + 1)) \land \text{WCP}(\Gamma)
\]

let \(n = \text{FAI}(q + \text{numreaden}.0)\) \/* Fetch and increment by 0 */
\[
\Gamma.\text{ctok} : \{N\} \times \{N \setminus 0, \ldots, v_1 + 1\} \land \Gamma.\text{history} : \text{Master}(H_2) \land \Gamma.wxtok : (0, \ldots, n - 1) \times (\text{dead}(H_2) \setminus \text{dead}(H_1))
\]

\[
\text{AccessList}(\Gamma, H_2, N) \land \forall tid < N. \cdot q + \text{rcounters} + \text{tid} : = \cdot \text{RCP}(\Gamma, tid)
\]

\[
\text{rcuCollect}(q, n, \text{newgc})
\]

\[
\text{RevokedUpTo}(\Gamma, H_2) \land \Gamma.\text{history} : \text{Master}(H_2) \land \Gamma.rxtok : (0, \ldots, N) \times (\text{dead}(H_2) \setminus \text{dead}(H_1))
\]

\[
\text{rcuDeallock}(q, x)
\]

\[
\exists H_1. \quad H_1 \leq H_3 \land \text{RevokedUpTo}(\Gamma, H_1) \land \text{DeallocBetween}(\Gamma, H_1, H_3)
\]

\[
\exists H_1. \quad H_1 \leq H_3 \land \text{SnapshotValid}(\Gamma, H_3) \land \exists i \in \text{dead}(H_3) \setminus \text{dead}(H_2) \land \forall n \leq N. \text{PermX}(\Gamma, x, n, i)
\]

C.4.8 Verification of \text{rcuDeallock}

\[
\Gamma.q = q \land H_1 \leq H_2 \leq H_3 \land \text{RevokedUpTo}(\Gamma, H_1) \land \text{DeallocBetween}(\Gamma, H_1, H_2)
\]

\[
\exists i \in \text{dead}(H_3) \setminus \text{dead}(H_2) \land \forall n \leq N. \text{PermX}(\Gamma, x, n, i)
\]

\[
\text{rcuDeallock}(q, x)
\]

\[
\exists H_1. \quad H_1 \leq H_3 \land \text{RevokedUpTo}(\Gamma, H_1) \land \text{DeallocBetween}(\Gamma, H_1, H_3)
\]

\[
\exists H_1. \quad H_1 \leq H_3 \land \text{SnapshotValid}(\Gamma, H_3) \land \exists i \in \text{dead}(H_3) \setminus \text{dead}(H_2) \land \forall n \leq N. \text{PermX}(\Gamma, x, n, i)
\]

\[
\text{RevokedUpTo}(\Gamma, H_1) \land \text{DeallocBetween}(\Gamma, H_1, H_2)
\]

\[
\text{push}(q + \text{del}, x)
\]
\[
\{ \text{RevokedUpTo}(\Gamma, H_1) \ast \text{DeallocationBetween}(\Gamma, H_1, H_3) \ast [\Gamma, \text{history} : \text{Master}(H_3)] \}
\]

let \( c = \text{choose}(1, 2) \)

if \( c == 1 \) then

0

\[
\{ \text{RevokedUpTo}(\Gamma, H_1) \ast \text{DeallocationBetween}(\Gamma, H_1, H_3) \ast [\Gamma, \text{history} : \text{Master}(H_3)] \ast \text{SnapshotValid}(\Gamma, H_3) \}
\]

else

\text{rcuSynchronize}(q)

\[
\{ \text{RevokedUpTo}(\Gamma, H_3) \ast \text{DeallocationBetween}(\Gamma, H_1, H_3) \ast [\Gamma, \text{history} : \text{Master}(H_3)] \}
\]

* \([\Gamma, \text{rxtok} : \{0, \ldots, N\} \times (\text{dead}(H_3) \setminus \text{dead}(H_1))]

\[
\exists L_d, L_d': \text{RevokedUpTo}(\Gamma, H_3) \ast \text{Stack}(q, L_d) \ast \Box(\text{NoDup}(L_d)) \ast \Box(\text{DeadFrom}(L_d, L_d', H_1, H_3))
\]

* \([\Gamma, \text{history} : \text{Master}(H_3)] \ast [\Gamma, \text{rxtok} : \{0, \ldots, N\} \times (\text{dead}(H_3) \setminus \text{dead}(H_1))]

\[
\text{RevokedUpTo}(\Gamma, H_3) \ast \text{Stack}(q, L_d) \ast [\Gamma, \text{history} : \text{Master}(H_3)] \ast [\Gamma, \text{rxtok} : \{0, \ldots, N\} \times (\text{dead}(H_3) \setminus \text{dead}(H_1))]
\]

\[
\exists L_1, L_2, L_1', L_2', L_1 \cdot L_2 = L_d \ast L_1' \cdot L_2' = L_d' \ast |L_2| = |L_2'|
\]

* \text{RevokedUpTo}(\Gamma, H_3) \ast [\Gamma, \text{rxtok} : \{0, \ldots, N\} \times ((\text{dead}(H_3) \setminus \text{dead}(H_1)) \setminus L_1') \ast \text{Stack}(q + \text{del}, L_2)

\]

repeat

\[
\Box(L_1 \cdot L_2 = L_d \ast L_1' \cdot L_2' = L_d' \ast |L_2| = |L_2'|) \ast \text{RevokedUpTo}(\Gamma, H_3) \ast [\Gamma, \text{history} : \text{Master}(H_3)]
\]

* \([\Gamma, \text{rxtok} : \{0, \ldots, N\} \times ((\text{dead}(H_3) \setminus \text{dead}(H_1)) \setminus L_1') \ast \text{Stack}(q + \text{del}, L_2)

\]

\[
\text{let } p = \text{pop}(q + \text{del})
\]

\[
\text{RevokedUpTo}(\Gamma, H_3) \ast [\Gamma, \text{history} : \text{Master}(H_3)] \ast [\Gamma, \text{rxtok} : \{0, \ldots, N\} \times ((\text{dead}(H_3) \setminus \text{dead}(H_1)) \setminus L_1')
\]

* \((p = 0 \ast L_2 = \text{nil} \ast \text{Stack}(q + \text{del}, \text{nil}))

\[
\lor (p \neq 0 \ast \exists P_2, P_2', i'. \ L_2 = p \cdot P_2 \ast L_2' = i' \cdot P_2' \ast \text{Stack}(q + \text{del}, P_2)

\]

* \(i' \in (\text{dead}(H_3) \setminus \text{dead}(H_1) \setminus L_1')

\]

if \( p == 0 \) then 1
else

\[
\text{RevokedUpTo}(\Gamma, H_3) \ast [\Gamma, \text{history} : \text{Master}(H_3)] \ast [\Gamma, \text{rxtok} : \{0, \ldots, N\} \times ((\text{dead}(H_3) \setminus \text{dead}(H_1)) \setminus L_1')
\]

* \(\text{Stack}(q + \text{del}, P_2) \ast \Box(L_2 = p \cdot P_2 \ast L_2' = i' \cdot P_2') \ast \Box(i' \in (\text{dead}(H_3) \setminus \text{dead}(H_1) \setminus L_1'))

\]

\[
\text{RevokedUpTo}(\Gamma, H_3) \ast [\Gamma, \text{history} : \text{Master}(H_3)] \ast [\Gamma, \text{rxtok} : \{0, \ldots, N\} \times ((\text{dead}(H_3) \setminus \text{dead}(H_1)) \setminus L_1' \cdot i')
\]

* \(\text{Stack}(q + \text{del}, P_2) \ast p + \text{data} \rightarrow = p + \text{link} \rightarrow i'

\]

\text{free}(p + \text{data});

\text{free}(p + \text{link});

\[
\text{RevokedUpTo}(\Gamma, H_3) \ast [\Gamma, \text{history} : \text{Master}(H_3)] \ast [\Gamma, \text{rxtok} : \{0, \ldots, N\} \times ((\text{dead}(H_3) \setminus \text{dead}(H_1)) \setminus L_1' \cdot i')
\]

* \(\text{Stack}(q + \text{del}, P_2)

\]

0

end

\[
\{ \text{RevokedUpTo}(\Gamma, H_3) \ast [\Gamma, \text{history} : \text{Master}(H_3)] \ast [\Gamma, \text{rxtok} : \{0, \ldots, N\} \times (\text{dead}(H_3) \setminus \text{dead}(H_1))]
\]

* \(\text{Stack}(q + \text{del}, \text{nil})

\]

32

2015/2/3
\[ \exists H_1, H_1 \leq H_3 \ast \text{RevokedUpTo}(\Gamma, H_1) \ast \text{Deallocation}(\Gamma, H_1, H_3) \]

\[ \ast \text{SnapshotValid}(\Gamma, H_3) \ast [\Gamma.text : \text{Master}(H_3)] \]

C.4.9 Verification of \textit{rcuNodeUpdate}

\[ \{ \text{WriterSafe}'(q, L \ast (x + \text{link}) \ast L') \ast P(v) \} \]

\texttt{rcuNodeUpdate}(q, x, p, v)

\[ \{ x'. \text{WriterSafe}'(q, L \ast (x' + \text{link}) \ast L') \} \]

This is one of the functions that does more than just swap around permissions via exchanges, so we take the time to give an intuitive sketch, as we tried to do for \textit{rcuInspectList}

Notice that the preconditions guarantee that the node will be in the liveset of the master trace, so the writer will be able to access the \( x + \text{link} \) and \( x + \text{data} \) fields, because \textit{WriterSafe}' contains \textit{AccessList} permissions.

The thread also needs to access the parent’s pointer (\( p \)). There are two cases. Either \( p \) is just another node in the list’s pointer, or \( p \) is \( x + \text{link} \), so that we’re changing the head of the list. These two cases are not treated differently in the code, but they do affect the permissions we use to be able to modify \( p \). In the former, the fact that the parent must point to the child node implies that it’s not dead, and hence the writer has the tokens to exchange for a permission to modify it. In the latter, the permission is passed in as part of the precondition.

At some point the writer needs to introduce a new abstract location, which will be paired with the new node’s \textit{link} pointer. We will pick one from the set of abstract locations that are not mentioned anywhere in the master trace. Since the set of nodes mentioned in the master trace is finite, the rule for introducing logical locations allows this.

The \textit{SinglePtr} property is important for the next step, which involves switching the old node to the dead state in the trace after we update the parent’s pointer. We can only do this if we show that no node is pointing to the old node any longer. But by the \textit{SinglePtr} property, the only node pointing it initially was the parent node, which we’ve now updated to point to the new node. Moreover, the new node we just added in does not point to it, because it points to the old node’s child, which again by the \textit{SinglePtr} cannot be the old node. So, the old node can be safely marked as dead.

Notice that it is only the writer that cares about the fact that there are no cycles in the list. The readers’ correctness does not rely on this directly. In fact, it would be fine for the writer to make a cycle in the list, the problem is that it would need to be careful about how it then updated the list if it did, since a node could have more than one parent, so simply changing one of the parent’s pointers would not be enough to make the node inaccessible.

\[ \{ \text{WriterSafe}'(q, L \ast (x + \text{link}) \ast L') \ast \square(P(v)) \} \]

\[ \exists \Gamma, H_1, H_2, H_3. \quad \frac{\Gamma.q = q \ast H_1 \leq H_2 \leq H_1 \ast \text{RevokedUpTo}(\Gamma, H_1) \ast \text{Deallocation}(\Gamma, H_1, H_2)}{\ast \text{InitialState}(\Gamma, L \ast (x + \text{link}) \ast L') \ast L_3} \]

\[ \square(\Gamma.q = q) \ast \square(H_1 \leq H_2 \leq H_1) \ast \text{RevokedUpTo}(\Gamma, H_1) \ast \text{Deallocation}(\Gamma, H_1, H_2) \]

\[ \ast \text{InitialState}(\Gamma, L \ast (x + \text{link}) \ast L') \ast L_3 \]

\[ \text{RevokedUpTo}(\Gamma, H_1) \ast \text{Deallocation}(\Gamma, H_1, H_2) \ast \text{InitialState}(\Gamma, L \ast (x + \text{link}) \ast L') \ast L_3 \]

\[ \ast \exists i_p, i_x. H_3(i_p) = i_x \ast \ast \forall n \leq N. \text{PermX}(\Gamma, n, i_x) \]

\[ \ast \left( \forall n \leq N. \text{PermX}(\Gamma, p, n, i_p) \right) \vee (p + \text{link} = q + \text{link} \ast \exists \epsilon \in [0,1]. q + \text{link} \not\sim i_p) \]

\[ \text{RevokedUpTo}(\Gamma, H_1) \ast \text{Deallocation}(\Gamma, H_1, H_2) \ast \text{InitialState}(\Gamma, L \ast (x + \text{link}) \ast L') \ast L_3 \]

\[ \ast \square(\exists i_p, i_x. H_3(i_p) = i_x) \ast \square(\forall n \leq N. \text{PermX}(\Gamma, n, i_x)) \]

\[ \ast \square(\left( \forall n \leq N. \text{PermX}(\Gamma, p, n, i_p) \right) \vee (p + \text{link} = q + \text{link} \ast \exists \epsilon \in [0,1]. q + \text{link} \not\sim i_p)) \]

let \( c = [x + \text{link}] \) at

\[ / \ast \text{The final disjunction below is pure but cannot be rendered as such due to its length} / \ast \]

\[ \text{RevokedUpTo}(\Gamma, H_1) \ast \text{Deallocation}(\Gamma, H_1, H_2) \ast \text{InitialState}(\Gamma, L \ast (x + \text{link}) \ast L') \ast L_3 \]

\[ \ast (c = 0 \ast (h_3(i_x)) = \text{null} \vee H_3(i_x) = \text{null}) \ast L' = \text{null} \]

\[ \vee (\exists i_c, L'''. \quad h_3(i_x)) = i_c \ast (\forall \text{tid} < N. \text{PermX}(\Gamma, c, \text{tid}, i_c)) \ast L' = c \ast L''' \]
let $x' = \text{alloc}(2)$ /* node = value, child pointer */

\begin{align*}
\{ & \text{RevokedUpTo}(\Gamma, H_1) \ast \text{DeallocBetween}(\Gamma, H_1, H_2) \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link} \cdot (x + \text{link}) \cdot L', H_3) \} \\
& \ast \text{uninit}(x' + \text{data}) \ast \text{uninit}(x' + \text{link}) \\
\{ & \text{RevokedUpTo}(\Gamma, H_1) \ast \text{DeallocBetween}(\Gamma, H_1, H_2) \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link} \cdot (x + \text{link}) \cdot L', H_3) \\
& \ast \text{uninit}(x' + \text{data}) \ast \text{uninit}(x' + \text{link}) \ast \exists i'_x, \text{uninit}(i'_x) \\
\{ & \text{RevokedUpTo}(\Gamma, H_1) \ast \text{DeallocBetween}(\Gamma, H_1, H_2) \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link} \cdot (x + \text{link}) \cdot L', H_3) \\
& \ast \text{uninit}(x' + \text{data}) \ast \text{uninit}(x' + \text{link}) \ast \text{uninit}(i'_x) \ast \Box(i'_x \notin \text{dom}(H_3)) \\
\} \\
\end{align*}

\begin{align*}
[x' + \text{data}]_{\text{sc}} & := v; \\
\{ & \text{RevokedUpTo}(\Gamma, H_1) \ast \text{DeallocBetween}(\Gamma, H_1, H_2) \ast \{ \Gamma, \text{history} : \text{Master}(H_3) \} \\
& \ast \Box(\text{SnapshotValid}(H_3)) \ast \Box(\text{TraceSpine}(\Gamma, H_3, L)) \ast \Box(\text{SinglePtr}(H_3)) \\
& \ast \text{uninit}(x' + \text{data}) \mapsto v \ast \text{uninit}(x' + \text{link}) \ast \text{uninit}(i'_x) \\
\} \\
\end{align*}

\begin{align*}
[x' + \text{link}]_{\text{at}} & := c; \\
\{ & \text{RevokedUpTo}(\Gamma, H_1) \ast \text{DeallocBetween}(\Gamma, H_1, H_2) \ast \{ \Gamma, \text{history} : \text{Master}(H_3 \cdot \text{upd}(i''_x, i_c)) \} \\
& \ast \text{SnapshotValid}(H_3 \cdot \text{upd}(i''_x, i_c)) \ast x' + \text{data} \mapsto v \ast x' + \text{link} \mapsto i'_x \\
& \ast (c = 0 \ast (\text{hd}(H_3(i_x)) = \text{null} \lor H_3(i_x) = \text{nil}) \ast L' = \text{nil} \ast [i'_x : \text{null} : \text{nil}] \text{LLP}(\Gamma, i'_x) \\
& \lor ((3i_c, L'', \text{hd}(H_3(i_x))) = i_c \ast (\forall \text{tid} < N. \text{PermX}(\Gamma, c, \text{tid}, i_c)) \ast L' = c \cdot L'' \ast [i'_x : i_c : \text{nil}] \text{LLP}(\Gamma, i'_x) \\
\} \\
\end{align*}

\begin{align*}
[p]_{\text{at}} & := x'; \\
\{ & \text{RevokedUpTo}(\Gamma, H_1) \ast \text{DeallocBetween}(\Gamma, H_1, H_2) \\
& \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link} \cdot (x' + \text{link}) \cdot L', H_3 \cdot \text{upd}(i''_x, i_c) \cdot \text{upd}(i_p, i'_p) \cdot \text{del}(i_x)) \\
\} \\
\end{align*}

rcuDealloc$(q, x)$;

\begin{align*}
\{ & \exists H_1, H_2 \leq (H_3 \cdot \text{upd}(i''_x, i_c) \cdot \text{upd}(i_p, i'_p) \cdot \text{del}(i_x)) \ast \text{RevokedUpTo}(\Gamma, H_1) \\
& \ast \text{DeallocBetween}(\Gamma, H_1, H_3 \cdot \text{upd}(i''_x, i_c) \cdot \text{upd}(i_p, i'_p) \cdot \text{del}(i_x)) \\
& \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link} \cdot (x' + \text{link}) \cdot L', H_3 \cdot \text{upd}(i''_x, i_c) \cdot \text{upd}(i_p, i'_p) \cdot \text{del}(i_x)) \\
\} \\
\end{align*}

\begin{align*}
\{ & x'. \text{WriterSafe}(q, L \cdot p + \text{link} \cdot (x' + \text{link}) \cdot L') \\
\} \\
\end{align*}

C.4.10 Verification of rcuNodeAppend

\begin{align*}
\{ & \text{WriterSafe}(q, L \cdot p + \text{link}) \ast P(v) \\
& \text{rcuNodeAppend}(q, p, v) \\
& \{ x. \text{WriterSafe}(q, L \cdot p + \text{link} \cdot (x + \text{link})) \} \\
\} \\
\end{align*}

This is similar to rcuNodeUpdate, but a bit simpler. Again, there are two cases in the precondition, based on whether $p$ is $q + \text{link}$ or the link field of a node.

\begin{align*}
\{ & \text{WriterSafe}(q, L \cdot p) \ast P(v) \\
\} \\
\end{align*}

\begin{align*}
\{ & \exists \Gamma, H_1, H_2, H_3. \Gamma.q = q \ast H_1 \leq H_2 \leq H_1 \ast \text{RevokedUpTo}(\Gamma, H_1) \ast \text{DeallocBetween}(\Gamma, H_1, H_2) \\
& \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link}, H_3) \\
\} \\
\end{align*}

\begin{align*}
\{ & \exists \Gamma. q = q \ast \Box(H_1 \leq H_2 \leq H_1) \ast \text{RevokedUpTo}(\Gamma, H_1) \ast \text{DeallocBetween}(\Gamma, H_1, H_2) \\
& \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link}, H_3) \\
\} \\
\end{align*}
\[
\begin{align*}
&\text{let } x = \text{alloc}(2) \quad /* \text{ node = value, child pointer */} \\
&\text{RevokedUpTo}(\Gamma, \Gamma_1) \ast \text{DeallocBetween}(\Gamma, \Gamma_1, \Gamma_2) \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link}, H_3) \\
&\quad \ast \exists_{p_i}. \left( (\forall n \leq N. \text{PermX}(\Gamma, p, n, i_p)) \lor (p + \text{link} = q + \text{link} \ast \exists \epsilon \in [0, 1]. q + \text{link} \xrightarrow{\epsilon} i_p) \right)
\end{align*}
\]

let \( x = \text{alloc}(2) \) /* node = value, child pointer */

\[
\begin{align*}
&\text{RevokedUpTo}(\Gamma, \Gamma_1) \ast \text{DeallocBetween}(\Gamma, \Gamma_1, \Gamma_2) \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link}, H_3) \\
&\quad \ast \text{uninit}(x + \text{data}) \ast \text{uninit}(x + \text{link})
\end{align*}
\]

\[
\begin{align*}
&\text{RevokedUpTo}(\Gamma, \Gamma_1) \ast \text{DeallocBetween}(\Gamma, \Gamma_1, \Gamma_2) \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link}, H_3) \\
&\quad \ast \text{uninit}(x + \text{data}) \ast \text{uninit}(x + \text{link}) \ast \exists_{x} \text{uninit}(i_x)
\end{align*}
\]

\[
\begin{align*}
&\text{RevokedUpTo}(\Gamma, \Gamma_1) \ast \text{DeallocBetween}(\Gamma, \Gamma_1, \Gamma_2) \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link}, H_3) \\
&\quad \ast \text{uninit}(x + \text{data}) \ast \text{uninit}(x + \text{link}) \ast \text{uninit}(i_x) \ast \square(i_x \notin \text{dom}(H_3))
\end{align*}
\]

\[
\begin{align*}
[x + \text{data}]_{\text{at}} := v; \\
&\text{RevokedUpTo}(\Gamma, \Gamma_1) \ast \text{DeallocBetween}(\Gamma, \Gamma_1, \Gamma_2) \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link}, H_3) \\
&\quad \ast x + \text{data} \rightarrow v \ast \text{uninit}(x + \text{link}) \ast \text{uninit}(i_x)
\end{align*}
\]

\[
\begin{align*}
[x + \text{link}]_{\text{at}} := 0; \\
&\text{RevokedUpTo}(\Gamma, \Gamma_1) \ast \text{DeallocBetween}(\Gamma, \Gamma_1, \Gamma_2) \ast [\Gamma.\text{history} : \text{Master}(H_3 \cdot \text{upd}(i_x, \text{null}))]; \\
&\quad \ast \text{SnapshotValid}(H_3 \cdot \text{upd}(i_x, \text{null})) \ast x + \text{data} \rightarrow v \ast x + \text{link} \rightarrow i_x \ast \square \left( i_x : \text{null} \ast \text{LLP}(\Gamma, i_x) \right)
\end{align*}
\]

\[
\begin{align*}
&\text{RevokedUpTo}(\Gamma, \Gamma_1) \ast \text{DeallocBetween}(\Gamma, \Gamma_1, \Gamma_2) \ast [\Gamma.\text{history} : \text{Master}(H_3 \cdot \text{upd}(i_x, \text{null}))]; \\
&\quad \ast \text{SnapshotValid}(H_3 \cdot \text{upd}(i_x, \text{null})) \ast \square(\text{vtid} \leq N. \text{PermX}(\Gamma, x, \text{tid}, i_x))
\end{align*}
\]

\[
\begin{align*}
[p]_{\text{at}} := x; \\
&\text{RevokedUpTo}(\Gamma, \Gamma_1) \ast \text{DeallocBetween}(\Gamma, \Gamma_1, \Gamma_2) \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link} \cdot (x + \text{link}), H_3 \cdot \text{upd}(i_x, \text{null}) \cdot \text{upd}(i_p, i_x))
\end{align*}
\]

\[
\begin{align*}
&\{ x. \text{WriterSafe}'(q, L \cdot p + \text{link} \cdot (x + \text{link})) \}
\end{align*}
\]

C.4.11 Verification of rcuNodeDel

\[
\begin{align*}
&\{ \text{WriterSafe}'(q, L \cdot (p + \text{link}) \cdot (x + \text{link}) \cdot L') \}
\end{align*}
\]

rcuNodeDel(q, x, p)

\[
\begin{align*}
&\{ \text{WriterSafe}'(q, L \cdot (p + \text{link}) \cdot L') \}
\end{align*}
\]

\[
\begin{align*}
&\exists \Gamma, H_1, H_2, H_3, \Gamma.q = q \ast H_1 \leq H_2 \leq H_1 \ast \text{RevokedUpTo}(\Gamma, \Gamma_1) \ast \text{DeallocBetween}(\Gamma, \Gamma_1, \Gamma_2) \\
&\quad \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link} \cdot (x + \text{link}) \cdot L'), H_3)
\end{align*}
\]

\[
\begin{align*}
&\square(\Gamma.q = q \ast \square(\Gamma.H_1 \leq H_2 \leq H_1) \ast \text{RevokedUpTo}(\Gamma, \Gamma_1) \ast \text{DeallocBetween}(\Gamma, \Gamma_1, \Gamma_2) \\
&\quad \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link} \cdot (x + \text{link} \cdot L'), H_3)
\end{align*}
\]

\[
\begin{align*}
&\text{RevokedUpTo}(\Gamma, \Gamma_1) \ast \text{DeallocBetween}(\Gamma, \Gamma_1, \Gamma_2) \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link} \cdot (x + \text{link} \cdot L'), H_3) \\
&\quad \ast \exists_{p_i, x}. H_3(i_p) = i_x \ast \ast \forall n \leq N. \text{PermX}(\Gamma, x, n, i_x)
\end{align*}
\]

\[
\begin{align*}
&\ast \left( (\forall n \leq N. \text{PermX}(\Gamma, p, n, i_p)) \lor (p + \text{link} = q + \text{link} \ast \exists \epsilon \in [0, 1]. q + \text{link} \xrightarrow{\epsilon} i_p) \right)
\end{align*}
\]

\[
\begin{align*}
&\text{RevokedUpTo}(\Gamma, \Gamma_1) \ast \text{DeallocBetween}(\Gamma, \Gamma_1, \Gamma_2) \ast \text{CurrentState}(\Gamma, L \cdot p + \text{link} \cdot (x + \text{link} \cdot L'), H_3) \\
&\quad \ast \square(\square H_3(i_p) = i_x \ast \ast \forall n \leq N. \text{PermX}(\Gamma, x, n, i_x))
\end{align*}
\]

\[
\begin{align*}
&\ast \left( (\forall n \leq N. \text{PermX}(\Gamma, p, n, i_p)) \lor (p + \text{link} = q + \text{link} \ast \exists \epsilon \in [0, 1]. q + \text{link} \xrightarrow{\epsilon} i_p) \right)
\end{align*}
\]

let \( c = [x + \text{link}]_{\text{at}} \)
// The final disjunction below is pure but cannot be rendered as such due to its length */

\[
\begin{align*}
\text{RevokedUpTo}(\Gamma, H_1) \land \text{DealocBetween}(\Gamma, H_1, H_2) \land \text{CurrentState}(\Gamma, L \cdot p + \text{link} \cdot (x + \text{link}) \cdot L', H_3) \\
\quad = (c = 0 \land (H_3(i_x) = \text{null} \lor H_3(i_x) = \text{nil}) \land L' = \text{nil}) \\
\quad \lor (\exists i_c, L''. H_3(i_x) = i_c \land (\forall \text{tid} < N. \text{PermX}(\Gamma, c, \text{tid}, i_c) \land L' = c \cdot L''))
\end{align*}
\]

\[\text{if} \quad [p]_{\text{at}} := c;\]

\[
\begin{align*}
\text{RevokedUpTo}(\Gamma, H_1) \land \text{DealocBetween}(\Gamma, H_1, H_2) & \land [\Gamma.\text{history} : \text{Master}(H_3 \cdot \text{ upd}(i_p, i_c) \cdot \text{ del}(i_x))] \\
\quad \land \text{SnapshotValid}(H_3 \cdot \text{ upd}(i_p, i_c) \cdot \text{ del}(i_x)) \\
\quad \land (c = 0 \land (H_3(i_x) = \text{null} \lor H_3(i_x) = \text{nil}) \land L' = \text{nil} \cdot i_p : \text{null} \cdot H_3(i_p) \cdot \text{ LLP}(\Gamma, i_p)) \\
\quad \land (\exists i_c, L''. H_3(i_x) = i_c \land (\forall \text{tid} < N. \text{PermX}(\Gamma, c, \text{tid}, i_c) \land L' = c \cdot L'') \cdot i_p : i_c : H_3(i_p) \cdot \text{ LLP}(\Gamma, i_p))
\end{align*}
\]

\[
\begin{align*}
\text{rcuDealloc}(q, x); \\
\{\text{WriterSafe}'(q, L \cdot p + \text{link} \cdot L')\}
\end{align*}
\]

C.4.12 Verification of Writer \( p + \text{link} \) rule

\[
\begin{align*}
\{\text{WriterSafe}'(q, L \cdot (p + \text{link}) \cdot L') \land (L' \neq \text{nil})\} \\
\quad \rightarrow (p + \text{link})_{\text{acq}} \\
\quad \rightarrow \{p'. \text{WriterSafe}'(q, L \cdot (p + \text{link}) \cdot L') \land \exists L''. L' = (p' + \text{link}) \cdot L''\}
\end{align*}
\]

Notice that from SnapshotValid and the definition of WriterSafe, the writer knows that the abstract location for \( p + \text{link} \) should tie to some abstract location \( i \), for which \( H(i) = i' \cdot \text{ } \) — where there’s a PermX for \( i' \) that links it to \( p' \). When it actually reads \( p + \text{link} \), it cannot discover that \( i' \) is actually in some different state, since this would contradict the master. Now, suppose the value read from the pointer is \( p'' \). If \( p'' \neq p' \), then the writer has knowledge of PermX(\( \Gamma, p', N, i' \)) and PermX(\( \Gamma, p'', N, i'' \)), which implies false, since the writer has rxtok \( (N, i') \).