Safe Deferred Memory Reclamation with Types

Ismail Kuru[0000−0002−5796−2150] and Colin S. Gordon[0000−0002−9012−4490]

Drexel University
{ik335,csgordon}@drexel.edu

Abstract. Memory management in lock-free data structures remains a major challenge in concurrent programming. Design techniques including read-copy-update (RCU) and hazard pointers provide workable solutions, and are widely used to great effect. These techniques rely on the concept of a grace period: nodes that should be freed are not deallocated immediately, and all threads obey a protocol to ensure that the deallocating thread can detect when all possible readers have completed their use of the object. This provides an approach to safe deallocation, but only when these subtle protocols are implemented correctly.
We present a static type system to ensure correct use of RCU memory management: that nodes removed from a data structure are always scheduled for subsequent deallocation, and that nodes are scheduled for deallocation at most once. As part of our soundness proof, we give an abstract semantics for RCU memory management primitives which captures the fundamental properties of RCU. Our type system allows us to give the first proofs of memory safety for RCU linked list and binary search tree implementations without requiring full verification.

1 Introduction

For many workloads, lock-based synchronization – even fine-grained locking – has unsatisfactory performance. Often lock-free algorithms yield better performance, at the cost of more complex implementation and additional difficulty reasoning about the code. Much of this complexity is due to memory management: developers must reason about not only other threads violating local assumptions, but whether other threads are *finished accessing* nodes to deallocate. At the time a node is unlinked from a data structure, an unknown number of additional threads may have already been using the node, having read a pointer to it before it was unlinked in the heap.

A key insight for manageable solutions to this challenge is to recognize that just as in traditional garbage collection, the unlinked nodes need not be reclaimed immediately, but can instead be reclaimed later after some protocol finishes running. Hazard pointers [30] are the classic example: all threads actively collaborate on bookkeeping data structures to track who is using a certain reference. For structures with read-biased workloads, Read-Copy-Update (RCU) [24] provides an appealing alternative. The programming style resembles a combination of reader-writer locks and lock-free programming. Multiple concurrent readers perform minimal bookkeeping – often nothing they wouldn't already do. A single writer at a time runs in parallel with readers, performing additional work to track which readers may have observed a node they wish to deallocate. There are now RCU implementations of many common tree data structures [8,34,5,25,3,20], and RCU plays a key role in Linux kernel memory management [28].

However, RCU primitives remain non-trivial to use correctly: developers must ensure they release each node exactly once, from exactly one thread, *after* ensuring other threads are finished with the node in question. Model checking can be used to validate correctness of implementations for a mock client [22,7,18,1], but this does not guarantee correctness of arbitrary client code. Sophisticated verification logics can prove correctness of the RCU primitives and clients [16,12,33,23]. But these techniques require significant verification expertise to apply, and are specialized to individual data structures or implementations. One important reason for the sophistication in these logics stems from the complexity of the underlying memory reclamation model. However, Meyer and Wolff [29] show that a suitable abstraction enables separating verifying *correctness* of concurrent data structures from its underlying reclamation model under the assumption of *memory safety*, and study proofs of correctness assuming memory safety.

We propose a type system to ensure that RCU client code uses the RCU primitives safely, ensuring memory safety for concurrent data structures using RCU memory management. We do this in a general way, not assuming the client implements any specific data structure, only one satisfying some basic properties common to RCU data structures (such as having a *tree* memory footprint). In order to do this, we must also give a formal operational model of the RCU primitives that abstracts many implementations, without assuming a particular implementation of the RCU primitives. We describe our RCU semantics and type system, prove our type system sound against the model (which ensures memory is reclaimed correctly), and show the type system in action on two important RCU data structures.

Our contributions include:

- A general (abstract) operational model for RCU-based memory management
- A type system that ensures code uses RCU memory management correctly, which is signifiantly simpler than full-blown verification logics
- Demonstration of the type system on two examples: a linked-list based bag and a binary search tree
- A proof that the type system guarantees memory safety when using RCU primitives.

2 Background & Motivation

In this section, we recall the general concepts of read-copy-update concurrency. We use the RCU linked-list-based bag [26] from Figure 1 as a running example. It includes annotations for our type system, which will be explained in Section 4.2.

As with concrete RCU implementations, we assume threads operating on a structure are either performing read-only traversals of the structure — *reader threads* — or are performing an update — *writer threads* — similar to the use of

```
struct BagNode{
  int data;
  BagNode<rcuItr> Next;
}
BagNode<rcuRoot> head;
void add(int toAdd){
WriteBegin;
BagNode nw = new;
{$nw$: rcuFresh{}}
nw.data = toAdd;
{$head$: rcuRoot, $par$: undef, $cur$: undef}
BagNode<rcuItr> par,cur = head;
{$head$: rcuRoot, $par$: rcuItr$\epsilon${}}
{$cur$: rcuItr$\epsilon${}}
cur = par.Next;
{$cur$: rcuItr$Next${}}
{$par$: rcuItr$\epsilon${$Next \mapsto cur$}}
while(cur.Next != null){
  {$cur$: rcuItr$(Next)^k.Next${}}
  {$par$: rcuItr$(Next)^k${$Next \mapsto cur$}}
  par = cur;
  cur = par.Next;
  {$cur$: rcuItr$(Next)^k.Next.Next${}}
  {$par$: rcuItr$(Next)^k.Next${$Next \mapsto cur$}}
}
{$nw$: rcuFresh{}}
{$cur$: rcuItr$(Next)^k.Next${$Next \mapsto null$}}
{$par$: rcuItr$(Next)^k${$Next \mapsto cur$}}
nw.Next= null;
{$nw$: rcuFresh{$Next \mapsto null$}}
{$cur$: rcuItr$(Next)^k.Next${$Next \mapsto null$}}
cur.Next=nw;
{$nw$: rcuItr$(Next)^k.Next.Next${$Next \mapsto null$}}
{$cur$: rcuItr$(Next)^k.Next${$Next \mapsto nw$}}
WriteEnd;
}
```

```
void remove(int toDel){
WriteBegin;
{$head$: rcuRoot, $par$ : undef, $cur$: undef}
BagNode<rcuItr> par,cur = head;
{$head$: rcuRoot, $par$: rcuItr$\epsilon${}, $cur$: rcuItr$\epsilon${}}
cur = par.Next;
{$cur$: rcuItr$Next${}}
{$par$: rcuItr$\epsilon${$Next \mapsto cur$}}
while(cur.Next != null&&cur.data != toDel)
{
  {$cur$: rcuItr$(Next)^k.Next${}}
  {$par$: rcuItr$(Next)^k${$Next \mapsto cur$}}
  par = cur;
  cur = par.Next;
  {$cur$: rcuItr$(Next)^k.Next.Next${}}
  {$par$: rcuItr$(Next)^k.Next${$Next \mapsto cur$}}
}
{$nw$: rcuFresh{}}
{$par$: rcuItr$(Next)^k${$Next \mapsto cur$}}
{$cur$: rcuItr$(Next)^k.Next${}}
BagNode<rcuItr> curl = cur.Next;
{$cur$: rcuItr$(Next)^k.Next${$Next \mapsto curl$}}
{$curl$: rcuItr$(Next)^k.Next.Next${}}
par.Next = curl;
{$par$: rcuItr$(Next)^k${$Next \mapsto curl$}}
{$cur$: unlinked}
{$cur$: rcuItr$(Next)^k.Next${}}
SyncStart;
SyncStop;
{$cur$: freeable}
Free(cur);
{$cur$: undef}
WriteEnd;
}
```

Fig. 1: RCU client: singly linked list based bag implementation.

many-reader single-writer reader-writer locks.[1] It differs, however, in that readers may execute concurrently with the (single) writer.

This distinction, and some runtime bookkeeping associated with the read- and write-side critical sections, allow this model to determine at modest cost when a node unlinked by the writer can safely be reclaimed.

Figure 1 gives the code for adding and removing nodes from a bag. Type checking for all code, including membership queries for bag, can be found in our technical report [21]. Algorithmically, this code is nearly the same as any sequential implementation. There are only two differences. First, the read-side critical section in `member` is indicated by the use of `ReadBegin` and `ReadEnd`; the write-side critical section is between `WriteBegin` and `WriteEnd`. Second, rather than immediately reclaiming the memory for the unlinked node, `remove` calls `SyncStart` to begin a *grace period* — a wait for reader threads that may still hold references to unlinked nodes to finish their critical sections. `SyncStop` blocks

[1] RCU implementations supporting multiple concurrent writers exist [3], but are the minority.

execution of the writer thread until these readers exit their read critical section (via `ReadEnd`). These are the essential primitives for the implementation of an RCU data structure.

These six primitives together track a critical piece of information: which reader threads' critical sections overlapped the writer's. Implementing them efficiently is challenging [8], but possible. The Linux kernel for example finds ways to reuse existing task switch mechanisms for this tracking, so readers incur no additional overhead. The reader primitives are semantically straightforward – they atomically record the start, or completion, of a read-side critical section.

The more interesting primitives are the write-side primitives and memory reclamation. `WriteBegin` performs a (semantically) standard mutual exclusion with regard to other writers, so only one writer thread may modify the structure *or the writer structures used for grace periods.*

`SyncStart` and `SyncStop` implement *grace periods* [32]: a mechanism to wait for readers to finish with any nodes the writer may have unlinked. A grace period begins when a writer requests one, and finishes when all reader threads active *at the start of the grace period* have finished their current critical section. Any nodes a writer unlinks before a grace period are physically unlinked, but not logically unlinked until after one grace period.

An attentive reader might already realize that our usage of logical/physical unlinking is different than the one used in data-structures literature where typically a *logical deletion* (e.g., marking) is followed by a *physical deletion* (unlinking). Because all threads are forbidden from holding an interior reference into the data structure after leaving their critical sections, waiting for active readers to finish their critical sections ensures they are no longer using any nodes the writer unlinked prior to the grace period. This makes actually freeing an unlinked node after a grace period safe.

`SyncStart` conceptually takes a snapshot of all readers active when it is run. `SyncStop` then blocks until all those threads in the snapshot have finished at least one critical section. `SyncStop` does not wait for *all* readers to finish, and does not wait for all overlapping readers to simultaneously be out of critical sections.

To date, every description of RCU semantics, most centered around the notion of a grace period, has been given algorithmically, as a specific (efficient) implementation. While the implementation aspects are essential to real use, the lack of an abstract characterization makes judging the correctness of these implementations – or clients – difficult in general. In Section 3 we give formal *abstract*, *operational* semantics for RCU implementations – inefficient if implemented directly, but correct from a memory-safety and programming model perspective, and not tied to specific low-level RCU implementation details. To use these semantics or a concrete implementation correctly, client code must ensure:

– Reader threads never modify the structure
– No thread holds an interior pointer into the RCU structure across critical sections
– Unlinked nodes are always freed by the unlinking thread *after* the unlinking, *after* a grace period, and *inside* the critical section

- Nodes are freed at most once

In practice, RCU data structures typically ensure additional invariants to simplify the above, e.g.:

- The data structure is always a tree
- A writer thread unlinks or replaces only one node at a time.

and our type system in Section 4 guarantees these invariants.

3 Semantics

In this section, we outline the details of an abstract semantics for RCU implementations. It captures the core client-visible semantics of most RCU primitives, but not the implementation details required for efficiency [28]. In our semantics, shown in Figure 2, an abstract machine state, MState, contains:

- A stack s, of type $\mathsf{Var} \times \mathsf{TID} \rightharpoonup \mathsf{Loc}$
- A heap, h, of type $\mathsf{Loc} \times \mathsf{FName} \rightharpoonup \mathsf{Val}$
- A lock, l, of type $\mathsf{TID} \uplus \{\mathsf{unlocked}\}$
- A root location rt of type Loc
- A read set, R, of type $\mathcal{P}(\mathsf{TID})$ and
- A bounding set, B, of type $\mathcal{P}(\mathsf{TID})$

The lock l enforces mutual exclusion between write-side critical sections. The root location rt is the root of an RCU data structure. We model only a single global RCU data structure; the generalization to multiple structures is straightforward but complicates formal development later in the paper. The reader set R tracks the thread IDs (TIDs) of all threads currently executing a read block. The bounding set B tracks which threads the writer is *actively* waiting for during a grace period — it is empty if the writer is not waiting.

Figure 2 gives operational semantics for *atomic* actions; conditionals, loops, and sequencing all have standard semantics, and parallel composition uses sequentially-consistent interleaving semantics.

The first few atomic actions, for writing and reading fields, assigning among local variables, and allocating new objects, are typical of formal semantics for heaps and mutable local variables. `Free` is similarly standard. A writer thread's critical section is bounded by `WriteBegin` and `WriteEnd`, which acquire and release the lock that enforces mutual exclusion between writers. `WriteBegin` only reduces (acquires) if the lock is $\mathsf{unlocked}$.

Standard RCU APIs include a primitive `synchronize_rcu()` to wait for a grace period for the current readers. We decompose this here into two actions, `SyncStart` and `SyncStop`. `SyncStart` initializes the blocking set to the current set of readers — the threads that may have already observed any nodes the writer has unlinked. `SyncStop` blocks until the blocking set is emptied by completing reader threads. However, it does not wait for *all* readers to finish, and does not

$$\alpha ::= \mathsf{skip} \mid \mathsf{x.f} = \mathsf{y} \mid \mathsf{y} = \mathsf{x} \mid \mathsf{y} = \mathsf{x.f} \mid \mathsf{y} = \mathsf{new} \mid \mathsf{Free(x)} \mid \mathsf{Sync} \qquad \mathsf{Sync} \overset{\Delta}{=} \mathsf{SyncStart};\mathsf{SyncStop}$$

$$\begin{array}{rrll}
(\textsc{RCU-WBegin}) & [\![\texttt{WriteBegin}]\!]\ (s,h,\mathsf{unlocked},rt,R,B) & \Downarrow_{tid}(s,h,l,rt,R,B) & \\
(\textsc{RCU-WEnd}) & [\![\texttt{WriteEnd}]\!]\ (s,h,l,rt,R,B) & \Downarrow_{tid}(s,h,\mathsf{unlocked},rt,R,B) & \\
(\textsc{RCU-RBegin}) & [\![\texttt{ReadBegin}]\!]\ (s,h,tid,rt,R,B) & \Downarrow_{tid}(s,h,tid,rt,R \uplus \{tid\},B) & tid \neq l \\
(\textsc{RCU-REnd}) & [\![\texttt{ReadEnd}]\!]\ (s,h,tid,rt,R \uplus \{tid\},B) & \Downarrow_{tid}(s,h,l,rt,R,B \setminus \{tid\}) & tid \neq l \\
(\textsc{RCU-SStart}) & [\![\texttt{SyncStart}]\!]\ (s,h,l,rt,R,\emptyset) & \Downarrow_{tid}(s,h,l,rt,R,R) & \\
(\textsc{RCU-SStop}) & [\![\texttt{SyncStop}]\!]\ (s,h,l,rt,R,\emptyset) & \Downarrow_{tid}(s,h,l,rt,R,\emptyset) & \\
(\textsc{Free}) & [\![\texttt{Free}(x)]\!]\ (s,h,l,rt,R,\emptyset) & \Downarrow_{tid}(s,h',l,rt,R,\emptyset) &
\end{array}$$

$$\text{provided } \forall_{f,o'}.\, rt \neq s(x,tid) \text{ and } o' \neq s(x,tid) \implies h(o',f) = h'(o',f) \text{ and } \forall_{f}.\, h'(o,f) = \mathsf{undef}$$

$$\begin{array}{rr}
(\textsc{HUpdt}) & [\![\texttt{x.f=y}]\!]\ (s,h,l,rt,R,B)\Downarrow_{tid}(s,h[s(x,tid),f \mapsto s(y,tid)],l,rt,R,B) \\
(\textsc{HRead}) & [\![\texttt{y=x.f}]\!]\ (s,h,l,rt,R,B)\Downarrow_{tid}(s[(y,tid) \mapsto h(s(x,tid),f)],h,l,rt,R,B) \\
(\textsc{SUpdt}) & [\![\texttt{y=x}]\!]\ (s,h,l,rt,R,B)\Downarrow_{tid}(s[(y,tid) \mapsto (x,tid)],h,l,rt,R,B) \\
(\textsc{HAlloc}) & [\![\texttt{y=new}]\!]\ (s,h,l,rt,R,B)\Downarrow_{tid}(s,h[\ell \mapsto \mathsf{nullmap}],l,rt,R,B)
\end{array}$$

$$\text{provided } rt \neq s(y,tid) \text{ and } s[(y,tid) \mapsto \ell], \text{ and}$$
$$h[\ell \mapsto \mathsf{nullmap}] \overset{\text{def}}{=} \lambda(o',f).\ \text{if } o = o' \text{ then } \mathit{skip} \text{ else } h(o',f)$$

Fig. 2: Operational semantics for RCU.

wait for all overlapping readers to simultaneously be out of critical sections. If two reader threads A and B overlap some SyncStart-SyncStop's critical section, it is possible that A may exit and re-enter a read-side critical section before B exits, and vice versa. Implementations must distinguish subsequent read-side critical sections from earlier ones that overlapped the writer's initial request to wait: since SyncStart is used *after* a node is physically removed from the data structure and readers may not retain RCU references across critical sections, A re-entering a fresh read-side critical section will not permit it to re-observe the node to be freed.

Reader thread critical sections are bounded by ReadBegin and ReadEnd. ReadBegin simply records the current thread's presence as an active reader. ReadEnd removes the current thread from the set of active readers, and also removes it (if present) from the blocking set — if a writer was waiting for a certain reader to finish its critical section, this ensures the writer no longer waits once that reader has finished its current read-side critical section.

Grace periods are implemented by the combination of ReadBegin, ReadEnd, SyncStart, and SyncStop. ReadBegin ensures the set of active readers is known. When a grace period is required, SyncStart;SyncStop; will store (in B) the active readers (which may have observed nodes before they were unlinked), and wait for reader threads to record when they have completed their critical section (and implicitly, dropped any references to nodes the writer wants to free) via ReadEnd.

These semantics do permit a reader in the blocking set to finish its read-side critical section and enter a *new* read-side critical section before the writer wakes. In this case, *the writer waits only for the first critical section of that reader to complete*, since entering the new critical section adds the thread's ID back to R, but not B.

4 Type System & Programming Language

In this section, we present a simple imperative programming language with two block constructs for modeling RCU, and a type system that ensures proper (memory-safe) use of the language. The type system ensures memory safety by enforcing these sufficient conditions:

- A heap node can only be freed if it is no longer accessible from an RCU data structure or from local variables of other threads. To achieve this we ensure the reachability and access which can be suitably restricted. We explain how our types support a delayed ownership transfer for the deallocation.
- Local variables may not point inside an RCU data structure unless they are inside an RCU read or write block.
- Heap mutations are *local*: each unlinks or replaces exactly one node.
- The RCU data structure remains a tree. While not a fundamental constraint of RCU, it is a common constraint across known RCU data structures because it simplifies reasoning (by developers or a type system) about when a node has become unreachable in the heap.

We also demonstrate that the type system is not only sound, but useful: we show how it types Figure 1's list-based bag implementation [26]. We also give type checked fragments of a binary search tree to motivate advanced features of the type system; the full typing derivation can be found in our technical report [21] Appendix B. The BST requires type narrowing operations that refine a type based on dynamic checks (e.g., determining which of several fields links to a node). In our system, we presume all objects contain all fields, but the number of fields is finite (and in our examples, small). This avoids additional overhead from tracking well-established aspects of the type system — class and field types and presence, for example — and focus on checking correct use of RCU primitives. Essentially, we assume the code our type system applies to is already type-correct for a system like C or Java's type system.

4.1 RCU Type System for Write Critical Section

Section 4.1 introduces RCU types and the need for subtyping. Section 4.2, shows how types describe program states, through code for Figure 1's list-based bag example. Section 4.3 introduces the type system itself.

RCU Types There are six types used in Write critical sections

$$\tau ::= \mathsf{rcuItr}\ \rho\ \mathcal{N} \mid \mathsf{rcuFresh}\ \mathcal{N} \mid \mathsf{unlinked} \mid \mathsf{undef} \mid \mathsf{freeable} \mid \mathsf{rcuRoot}$$

rcuItr is the type given to references pointing into a shared RCU data structure. A rcuItr type can be used in either a write region or a read region (without the additional components). It indicates both that the reference points into the shared RCU data structure and that the heap location referenced by rcuItr reference is reachable by following the path ρ from the root. A component $\mathcal{N}$ is a set of field mappings taking the field name to local variable names. Field maps are extended when the referent's fields are read. The field map and path components track reachability from the root, and local reachability between nodes. These are used to ensure the structure remains acyclic, and for the type system to recognize exactly when unlinking can occur.

Read-side critical sections use rcultr without path or field map components. These components are both unnecessary for readers (who perform no updates) and would be invalidated by writer threads anyways. Under the assumption that reader threads do not hold references across critical sections, the read-side rules essentially only ensure the reader performs no writes, so we omit the reader critical section type rules. They can be found in our technical report [21] Appendix E.

unlinked is the type given to references to unlinked heap locations — objects previously part of the structure, but now unreachable via the heap. A heap location referenced by an unlinked reference may still be accessed by reader threads, which may have acquired their own references before the node became unreachable. Newly-arrived readers, however, will be unable to gain access to these referents.

freeable is the type given to references to an unlinked heap location that is safe to reclaim because it is known that no concurrent readers hold references to it. Unlinked references become freeable after a writer has waited for a full grace period.

undef is the type given to references where the content of the referenced location is inaccessible. A local variable of type freeable becomes undef after reclaiming that variable's referent.

rcuFresh is the type given to references to freshly allocated heap locations. Similar to rcultr type, it has field mappings set $\mathcal{N}$. We set the field mappings in the set of an existing rcuFresh reference to be the same as field mappings in the set of rcultr reference when we replace the heap referenced by rcultr with the heap referenced by rcuFresh for memory safe replacement.

rcuRoot is the type given to the fixed reference to the root of the RCU data structure. It may not be overwritten.

Subtyping It is sometimes necessary to use imprecise types — mostly for control flow joins. Our type system performs these abstractions via subtyping on individual types and full contexts, as in Figure 3.

Figure 3 includes four judgments for subtyping. The first two — $\vdash \mathcal{N} \prec: \mathcal{N}'$ and $\vdash \rho \prec: \rho'$ — describe relaxations of field maps and paths respectively. $\vdash \mathcal{N} \prec: \mathcal{N}'$ is read as "the field map $\mathcal{N}$ is more precise than $\mathcal{N}'$" and similarly for paths. The third judgment $\vdash T \prec: T'$ uses path and field map subtyping to give subtyping among rcultr types — one rcultr is a subtype of another if its paths and field maps are similarly more precise — and to allow rcultr references to be

$$\mathcal{N} = \{f_0|\ldots|f_n \rightharpoonup \{y\} \mid f_i \in \mathsf{FName} \wedge\ 0 \le i \le n \wedge\ (y \in \mathsf{Var} \vee y \in \{null\})\} \quad \mathcal{N}_{f,\emptyset} = \mathcal{N} \setminus \{f \rightharpoonup \_\}$$

$$\mathcal{N}_{\emptyset} = \{\} \quad \mathcal{N}(\cup_{f \rightharpoonup y}) = \mathcal{N} \cup \{f \rightharpoonup y\} \quad \mathcal{N}(\setminus_{f \rightharpoonup y}) = \mathcal{N} - \{f \rightharpoonup y\}$$

$$\mathcal{N}([f \rightharpoonup y]) = \mathcal{N} \text{ where } f \rightharpoonup y \in \mathcal{N} \quad \mathcal{N}(f \rightharpoonup x \setminus y) = \mathcal{N} \setminus \{f \rightharpoonup x\} \cup \{f \rightharpoonup y\}$$

$$\boxed{\vdash \mathcal{N} \prec: \mathcal{N}'} \qquad \frac{\text{(T-NSub3)}}{\vdash \mathcal{N}_{f,\emptyset} \prec: \mathcal{N}([f \rightharpoonup y])} \qquad \frac{\text{(T-NSub4)}}{\vdash \mathcal{N}_{\emptyset} \prec: \mathcal{N}} \qquad \frac{\text{(T-NSub5)}}{\vdash \mathcal{N} \prec: \mathcal{N}}$$

$$\text{(T-NSub2)}\ \frac{}{\vdash \mathcal{N}([f_2 \rightharpoonup y]) \prec: \mathcal{N}([f_1|f_2 \rightharpoonup y])} \qquad \text{(T-NSub1)}\ \frac{}{\vdash \mathcal{N}([f_1 \rightharpoonup y]) \prec: \mathcal{N}([f_1|f_2 \rightharpoonup y])}$$

$$\boxed{\vdash \rho \prec: \rho'} \quad \text{(T-PSub1)}\ \frac{}{\vdash \rho.f_1 \prec: \rho.f_1|f_2} \qquad \text{(T-PSub2)}\ \frac{}{\vdash \rho.f_2 \prec: \rho.f_1|f_2} \qquad \text{(T-PSub3)}\ \frac{}{\vdash \rho \prec: \rho}$$

$$\boxed{\vdash T \prec: T'} \qquad \frac{\text{(T-TSub2)}}{\vdash \mathsf{rcultr} \prec: \mathsf{rcultr}} \qquad \frac{\text{(T-TSub)}}{\vdash \mathsf{rcultr}\ \_ \prec: \mathsf{undef}} \qquad \frac{\substack{\text{(T-TSub1)} \\ \vdash \rho \prec: \rho' \qquad \vdash \mathcal{N} \prec: \mathcal{N}'}}{\vdash \mathsf{rcultr}\, \rho\, \mathcal{N} \prec: \mathsf{rcultr}\, \rho'\, \mathcal{N}'}$$

$$\boxed{\vdash \Gamma \prec: \Gamma'} \quad \text{(T-CSub1)}\ \frac{\vdash \Gamma \prec: \Gamma' \quad \vdash \mathrm{T} \prec: \mathrm{T'}}{\vdash \Gamma\, , x : \mathrm{T} \prec: \Gamma'\, , x : \mathrm{T'}} \qquad \text{(T-CSub)}\ \frac{}{\vdash \Gamma \prec: \Gamma}$$

Fig. 3: Subtyping rules.

$$\boxed{\Gamma \vdash_{M,R} C \dashv \Gamma'} \qquad \frac{\text{(T-ReIndex)}}{\Gamma \vdash C_k \dashv \Gamma[\rho.f^k/\rho.f^k.f]} \qquad \text{(T-Loop1)}\ \frac{\Gamma(x) = \mathsf{bool} \qquad \Gamma \vdash C \dashv \Gamma}{\Gamma \vdash \mathsf{while}(x)\{C\} \dashv \Gamma}$$

$$\text{(T-Branch1)}\ \frac{\Gamma\, , x : \mathsf{rcultr}\, \rho\, \mathcal{N}([f_1 \rightharpoonup z]) \vdash C_1 \dashv \Gamma_4 \qquad \Gamma\, , x : \mathsf{rcultr}\, \rho\, \mathcal{N}([f_2 \rightharpoonup z]) \vdash C_2 \dashv \Gamma_4}{\Gamma\, , x : \mathsf{rcultr}\, \rho\, \mathcal{N}([f_1 \mid f_2 \rightharpoonup z]) \vdash \mathsf{if}(x.f_1 == z)\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \dashv \Gamma_4}$$

$$\text{(T-Branch3)}\ \frac{\Gamma,\ x : \mathsf{rcultr}\, \rho\, \mathcal{N}([f \rightharpoonup y \setminus \mathsf{null}]) \vdash C_1 \dashv \Gamma' \qquad \Gamma,\ x : \mathsf{rcultr}\, \rho\, \mathcal{N}([f \rightharpoonup y]) \vdash C_2 \dashv \Gamma'}{\Gamma,\ x : \mathsf{rcultr}\, \rho\, \mathcal{N}([f \rightharpoonup y]) \vdash \mathsf{if}(x.f == \mathsf{null})\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \dashv \Gamma'}$$

$$\text{(T-Loop2)}\ \frac{\Gamma,\ x : \mathsf{rcultr}\ \rho\ \mathcal{N}([f \rightharpoonup \_]) \vdash C \dashv \Gamma, x : \mathsf{rcultr}\ \rho'\ \mathcal{N}([f \rightharpoonup \_])}{\Gamma,\ x : \mathsf{rcultr}\ \rho\ \mathcal{N}([f \rightharpoonup \_]) \vdash \mathsf{while}(x.f \neq \mathsf{null})\{C\} \dashv x : \mathsf{rcultr}\ \rho'\ \mathcal{N}([f \rightharpoonup \mathsf{null}]),\ \Gamma}$$

$$\text{(T-Branch2)}\ \frac{\Gamma(x) = \mathsf{bool} \qquad \Gamma \vdash C_1 \dashv \Gamma' \qquad \Gamma \vdash C_2 \dashv \Gamma'}{\Gamma \vdash \mathsf{if}(x)\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \dashv \Gamma'}$$

Fig. 4: Type rules for control-flow.

"forgotten" — this is occasionally needed to satisfy non-interference checks in the type rules. The final judgment $\vdash \Gamma \prec: \Gamma'$ extends subtyping to all assumptions in a type context.

It is often necessary to abstract the contents of field maps or paths, without simply forgetting the contents entirely. In a binary search tree, for example, it may be the case that one node is a child of another, but *which* parent field points to the child depends on which branch was followed in an earlier conditional (consider the lookup in a BST, which alternates between following left and right children). In Figure 5, we see that `cur` aliases different fields of `par` – either *Left* or *Right* – in different branches of the conditional. The types after the conditional must overapproximate this, here as $Left|Right \mapsto cur$ in `par`'s field map, and a similar path disjunction in `cur`'s path. This is reflected in Figure 3's T-NSub1-5 and T-PSub1-2 – within each branch, each type is coerced to a supertype to validate the control flow join.

Another type of control flow join is handling loop invariants – where paths entering the loop meet the back-edge from the end of a loop back to the start for repetition. Because our types include paths describing how they are reachable from the root, some abstraction is required to give loop invariants that work for any number of iterations – in a loop traversing a linked list, the iterator pointer would naïvely have different paths from the root on each iteration, so the exact path is not loop invariant. However, the paths explored by a loop are regular, so we can abstract the paths by permitting (implicitly) existentially quantified indexes on path fragments, which express the existence of *some* path, without saying *which* path. The use of an explicit abstract repetition allows the type system to preserve the fact that different references have common path prefixes, even after a loop.

Assertions for the `add` function in lines 19 and 20 of Figure 1 show the *loop*'s effects on paths of iterator references used inside the loop, `cur` and `par`. On line 20, `par`'s path contains has $(Next)^k$. The k in the $(Next)^k$ abstracts the number of loop iterations run, implicitly assumed to be non-negative. The trailing $Next$ in `cur`'s path on line 19 – $(Next)^k.Next$ – expresses the relationship between `cur` and `par`: `par` is reachable from the root by following $Next$ k times, and `cur` is reachable via one additional $Next$. The types of 19 and 20, however, are not the same as lines 23 and 24, so an additional adjustment is needed for the types to become loop-invariant. *Reindexing* (T-ReIndex in Figure 4) effectively increments an abstract loop counter, contracting $(Next)^k.Next$ to $Next^k$ everywhere in a type environment. This expresses the same relationship between `par` and `cur` as before the loop, but the choice of k to make these paths accurate after each iteration would be one larger than the choice before. Reindexing the type environment of lines 23–24 yields the type environment of lines 19–20, making the types loop invariant. The reindexing essentially chooses a new value for the abstract k. This is sound, because the uses of framing in the heap mutation related rules of the type system ensure uses of any indexing variable are never separated – either all are reindexed, or none are.

```
{cur : rcultr Left|Right {},  par : rcultr ε {Left|Right ↦ cur}}
if(par.Left == cur){
  {cur : rcultr Left {},  par : rcultr ε {Left ↦ cur}}
  par = cur;
  cur = par.Left;
  {cur : rcultr Left.Left {},  par : rcultr Left {Left ↦ cur}}
}else{
  {cur : rcultr Right {},  par : rcultr ε {Right ↦ cur}}
  par = cur;
  cur = par.Right;
  {cur : rcultr Right.Right {},  par : rcultr Right {Right ↦ cur}}
}
{cur : rcultr Left|Right.Left|Right {},  par : rcultr Left|Right {Left|Right ↦ cur}}
```

Fig. 5: Choosing fields to read.

While abstraction is required to deal with control flow joins, reasoning about whether and which nodes are unlinked or replaced, and whether cycles are created, requires precision. Thus the type system also includes means (Figure 4) to refine imprecise paths and field maps. In Figure 5, we see a conditional with the condition $par.Left == cur$. The type system matches this condition to the imprecise types in line 1's typing assertion, and refines the initial type assumptions in each branch accordingly (lines 2 and 7) based on whether execution reflects the truth or falsity of that check. Similarly, it is sometimes required to check – and later remember – whether a field is null, and the type system supports this.

4.2 Types in Action

The system has three forms of typing judgement: $\Gamma \vdash C$ for standard typing outside RCU critical sections; $\Gamma \vdash_R C \dashv \Gamma'$ for reader critical sections, and $\Gamma \vdash_M C \dashv \Gamma'$ for writer critical sections. The first two are straightforward, essentially preventing mutation of the data structure, and preventing nesting of a writer critical section inside a reader critical section. The last, for writer critical sections, is flow sensitive: the types of variables may differ before and after program statements. This is required in order to reason about local assumptions at different points in the program, such as recognizing that a certain action may unlink a node. Our presentation here focuses exclusively on the judgment for the write-side critical sections.

Below, we explain our types through the list-based bag implementation [26] from Figure 1, highlighting how the type rules handle different parts of the code. Figure 1 is annotated with "assertions" – local type environments – in the style of a Hoare logic proof outline. As with Hoare proof outlines, these annotations can be used to construct a proper typing derivation.

Reading a Global RCU Root All RCU data structures have fixed roots, which we characterize with the rcuRoot type. Each operation in Figure 1 begins by reading the root into a new rcuItr reference used to begin traversing the structure. After each initial read (line 12 of `add` and line 4 of `remove`), the path of `cur` reference is the empty path (ϵ) and the field map is empty ($\{\}$), because it is an alias to the root, and none of its field contents are known yet.

Reading an Object Field and a Variable As expected, we explore the heap of the data structure via reading the objects' fields. Consider line 6 of `remove` and its corresponding pre- and post- type environments. Initially `par`'s field map is empty. After the field read, its field map is updated to reflect that its $Next$ field is aliased in the local variable `cur`. Likewise, afer the update, `cur`'s path is $Next$ ($= \epsilon \cdot Next$), extending the par node's path by the field read. This introduces field aliasing information that can subsequently be used to reason about unlinking.

Unlinking Nodes Line 24 of `remove` in Figure 1 unlinks a node. The type annotations show that before that line `cur` is in the structure (rcuItr), while afterwards its type is unlinked. The type system checks that this unlink disconnects only one node: note how the types of `par`, `cur`, and `curl` just before line 24 completely describe a section of the list.

Grace and Reclamation After the referent of `cur` is unlinked, concurrent readers traversing the list may still hold references. So it is not safe to actually reclaim the memory until after a grace period. Lines 28–29 of `remove` initiate a grace period and wait for its completion. At the type level, this is reflected by the change of `cur`'s type from unlinked to freeable, reflecting the fact that the grace period extends until any reader critical sections that might have observed the node in the structure have completed. This matches the precondition required by our rules for calling `Free`, which further changes the type of `cur` to undef reflecting that `cur` is no longer a valid reference. The type system also ensures no local (writer) aliases exist to the freed node and understanding this enforcement is twofold. First, the type system requires that only unlinked heap nodes can be freed. Second, framing relations in rules related to the heap mutation ensure no local aliases still consider the node linked.

Fresh Nodes Some code must also allocate new nodes, and the type system must reason about how they are incorporated into the shared data structure. Line 8 of the add method allocates a new node `nw`, and lines 10 and 29 initialize its fields. The type system gives it a fresh type while tracking its field contents, until line 32 inserts it into the data structure. The type system checks that nodes previously reachable from `cur` remain reachable: note the field maps of `cur` and `nw` in lines 30–31 are equal (trivially, though in general the field need not be null).

4.3 Type Rules

Figure 6 gives the primary type rules used in checking write-side critical section code as in Figure 1.

T-Root reads a root pointer into an rcultr reference, and T-ReadS copies a local variable into another. In both cases, the free variable condition ensures that updating the modified variable does not invalidate field maps of other variables in Γ. These free variable conditions recur throughout the type system, and we will not comment on them further. T-Alloc and T-Free allocate and reclaim objects. These rules are relatively straightforward. T-ReadH reads a field into a local variable. As suggested earlier, this rule updates the post-environment to reflect that the overwritten variable z holds the same value as $x.f$. T-WriteFH updates a field of a *fresh* (thread-local) object, similarly tracking the update in the fresh object's field map at the type level. The remaining rules are a bit more involved, and form the heart of the type system.

Grace Periods T-Sync gives pre- and post-environments to the compound statement `SyncStart;SyncStop` implementing grace periods. As mentioned earlier, this updates the environment afterwards to reflect that any nodes unlinked before the wait become freeable afterwards.

Unlinking T-UnlinkH type checks heap updates that remove a node from the data structure. The rule assumes three objects x, z, and r, whose identities we will conflate with the local variable names in the type rule. The rule checks the case where $x.f_1 == z$ and $z.f_2 == r$ initially (reflected in the path and field map components, and a write $x.f_1 = r$ removes z from the data structure (we assume, and ensure, the structure is a tree).

$$\boxed{\Gamma \vdash_M \alpha \dashv \Gamma'} \quad (\textsc{T-Root})\ \frac{y \notin \mathsf{FV}(\Gamma)}{\Gamma, r{:}\mathsf{rcuRoot}, y{:}\mathsf{undef} \vdash y = r \dashv y{:}\mathsf{rcultr}\epsilon\mathcal{N}_\emptyset, r{:}\mathsf{rcuRoot}, \Gamma}$$

$$(\textsc{T-ReadS})\ \frac{z \notin \mathsf{FV}(\Gamma)}{\Gamma, z : \_, x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N} \vdash z = x \dashv x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}, z : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}, \Gamma}$$

$$(\textsc{T-Alloc})\ \frac{}{\Gamma, x{:}\mathsf{undef} \vdash x = \texttt{new} \dashv x{:}\mathsf{rcuFresh}\mathcal{N}_\emptyset, \Gamma} \quad (\textsc{T-Free})\ \frac{}{x{:}\mathsf{freeable} \vdash \mathsf{Free}(x) \dashv x{:}\mathsf{undef}}$$

$$(\textsc{T-ReadH})\ \frac{\rho.f = \rho' \qquad z \notin \mathsf{FV}(\Gamma)}{\Gamma, z : \_, x{:}\mathsf{rcultr}\rho\mathcal{N} \vdash z = x.f \dashv x{:}\mathsf{rcultr}\rho\mathcal{N}([f \rightharpoonup z]), z{:}\mathsf{rcultr}\rho'\mathcal{N}_\emptyset, \Gamma}$$

(T-WriteFH)

$$\frac{z : \mathsf{rcultr}\rho.f\_ \quad \mathcal{N}(f) = z \quad f \notin dom(\mathcal{N}')}{\Gamma, p{:}\mathsf{rcuFresh}\mathcal{N}', x{:}\mathsf{rcultr}\rho\mathcal{N} \vdash_M p.f = z \dashv p{:}\mathsf{rcuFresh}\mathcal{N}'([f \rightharpoonup z]), x{:}\mathsf{rcultr}\rho\mathcal{N}([f \rightharpoonup z]), \Gamma}$$

$$(\textsc{T-Sync})\ \frac{}{\Gamma \vdash \mathsf{SyncStart}; \mathsf{SyncStop} \dashv \Gamma[\overline{x{:}\mathsf{freeable}/x{:}\mathsf{unlinked}}]}$$

(T-UnlinkH)

$$\frac{\begin{array}{c} \mathcal{N}(f_1) = z \qquad \rho.f_1 = \rho_1 \qquad \rho_1.f_2 = \rho_2 \\ \mathcal{N}' = \mathcal{N}([f_1 \rightharpoonup z \setminus r]) \qquad \forall_{f \in dom(\mathcal{N}_1)}.\, f \neq f_2 \implies (\mathcal{N}_1(f) = \mathsf{null}) \qquad \mathcal{N}(f_1) = z \qquad \mathcal{N}_1(f_2) = r \\ \forall_{n \in \Gamma, m, \mathcal{N}_3, p_3, f}.\, n{:}\mathsf{rcultr}\,\rho_3\,\mathcal{N}_3([f \rightharpoonup m]) \implies \begin{cases} ((\neg\mathsf{MayAlias}(\rho_3, \{\rho, \rho_1, \rho_2\})) \wedge (m \notin \{z, r\})) \\ \wedge(\forall_{\rho_4 \neq \epsilon}.\, \neg\mathsf{MayAlias}(\rho_3, \rho_2.\rho_4)) \end{cases} \end{array}}{\Gamma, x{:}\mathsf{rcultr}\rho\mathcal{N}, z{:}\mathsf{rcultr}\rho_1\mathcal{N}_1, r{:}\mathsf{rcultr}\rho_2\mathcal{N}_2 \vdash x.f_1 = r \dashv z{:}\mathsf{unlinked}, x{:}\mathsf{rcultr}\rho\mathcal{N}', r{:}\mathsf{rcultr}\rho_1\mathcal{N}_2, \Gamma}$$

(T-Replace)

$$\frac{\begin{array}{c} \mathcal{N}(f) = o \qquad \mathcal{N}' = \mathcal{N}([f \rightharpoonup o \setminus n]) \qquad \rho.f = \rho_1 \qquad \mathcal{N}_1 = \mathcal{N}_2 \qquad \mathsf{FV}(\Gamma) \cap \{p, o, n\} = \emptyset \\ \forall_{x \in \Gamma, \mathcal{N}_3, \rho_2, f_1, y}.\, (x{:}\mathsf{rcultr}\,\rho_2\,\mathcal{N}_3([f_1 \rightharpoonup y])) \implies (\neg\mathsf{MayAlias}(\rho_2, \{\rho, \rho_1\}) \wedge (y \neq o)) \end{array}}{\Gamma, p{:}\mathsf{rcultr}\rho\mathcal{N}, o{:}\mathsf{rcultr}\rho_1\mathcal{N}_1, n{:}\mathsf{rcuFresh}\mathcal{N}_2 \vdash p.f = n \dashv p{:}\mathsf{rcultr}\rho\mathcal{N}', n{:}\mathsf{rcultr}\rho_1\mathcal{N}_2, o{:}\mathsf{unlinked}, \Gamma}$$

(T-Insert)

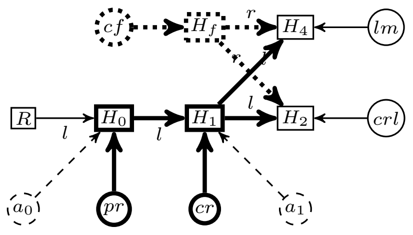
$$\frac{\begin{array}{c} \mathcal{N}' = \mathcal{N}([f \rightharpoonup o \setminus n]) \qquad \rho.f = \rho_1 \qquad \rho_1.f_4 = \rho_2 \\ \mathcal{N}(f) = \mathcal{N}_1(f_4) \qquad \forall_{f_2 \in dom(\mathcal{N}_1)}.\, f_4 \neq f_2 \implies \mathcal{N}_1(f_2) = \mathsf{null} \qquad \mathsf{FV}(\Gamma) \cap \{p, o, n\} = \emptyset \\ \forall_{x \in \Gamma, \mathcal{N}_3, \rho_3, f_1, y}.\, (x : \mathsf{rcultr}\,\rho_3\,\mathcal{N}_3([f_1 \rightharpoonup y])) \implies (\forall_{\rho_4 \neq \epsilon}.\, \neg\mathsf{MayAlias}(\rho_3, \rho.\rho_4)) \end{array}}{\Gamma, p{:}\mathsf{rcultr}\rho\mathcal{N}, o{:}\mathsf{rcultr}\rho_1\mathcal{N}_2, n{:}\mathsf{rcuFresh}\mathcal{N}_1 \vdash p.f = n \dashv p{:}\mathsf{rcultr}\rho\mathcal{N}', n{:}\mathsf{rcultr}\rho_1\mathcal{N}_1, o{:}\mathsf{rcultr}\rho_2\mathcal{N}_2, \Gamma}$$

$$\boxed{\Gamma \vdash_M C \dashv \Gamma'} \quad (\textsc{ToRCUWrite})\ \frac{\begin{array}{c} \mathsf{NoFresh}(\Gamma') \qquad \mathsf{NoUnlinked}(\Gamma') \qquad \mathsf{NoFreeable}(\Gamma') \\ \Gamma, y{:}\mathsf{rcultr}\_ \vdash_M C \dashv \Gamma' \qquad \mathsf{FType}(f) = \mathsf{RCU} \end{array}}{\Gamma \vdash \mathsf{RCUWrite}\, x.f\ \mathsf{as}\ y\ \mathsf{in}\ \{C\}}$$

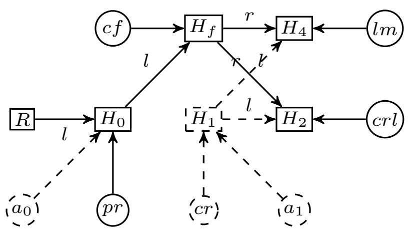
Fig. 6: Type rules for write side critical section.

The rule must also avoid unlinking multiple nodes: this is the purpose of the first (smaller) implication: it ensures that beyond the reference from z to r, all fields of z are null.

Finally, the rule must ensure that no types in Γ are invalidated. This could happen one of two ways: either a field map in Γ for an alias of x duplicates the assumption that $x.f_1 == z$ (which is changed by this write), or Γ contains a descendant of r, whose path from the root will change when its ancestor is modified. The final assumption of T-UnlinkH (the implication) checks that for every rcultr reference n in Γ, it is not a path alias of x, z, or r; no entry of its field map (m) refers to r or z (which would imply n aliased x or z initially); and its



(a) *Freshly* allocated heap node referenced by cf



(b) Safe replacement of the heap node referenced by cr with the *fresh* heap node referenced by cf.

Fig. 7: Replacing *existing* heap nodes with *fresh* ones. Type rule T-REPLACE.

path is not an extension of r (i.e., it is not a descendant). MayAlias is a predicate on two paths (or a path and set of paths) which is true if it is possible that any concrete paths the arguments may abstract (e.g., via adding non-determinism through | or abstracting iteration with indexing) *could* be the same. The negation of a MayAlias use is true only when the paths are guaranteed to refer to different locations in the heap.

Replacing with a Fresh Node Replacing with a rcuFresh reference faces the same aliasing complications as direct unlinking. We illustrate these challenges in Figures 7a and 7b. Our technical report [21] also includes Figures 32a and 32b in Appendix D to illustrate complexities in unlinking. The square R nodes are root nodes, and H nodes are general heap nodes. All resources in thick straight lines and dotted lines form the memory foot print of a node replacement. The hollow thick circular nodes – *pr* and *cr* – point to the nodes involved in replacing H_1 (referenced by `cr`) wih H_f (referenced by cf) in the structure. We may have a_0 and a_1 which are aliases with *pr* and *cr* respectively. They are *path-aliases* as they share the same path from root to the node that they reference. Edge labels l and r are abbreviations for the $Left$ and *Right* fields of a binary search tree. The thick dotted H_f denotes the freshly allocated heap node referenced by thick dotted cf. The thick dotted field l is set to point to the referent of *cl* and the thick dotted field r is set to point to the referent of the heap node referenced by lm.

H_f initially (Figure 7a) is not part of the shared structure. If it was, it would violate the tree shape requirement imposed by the type system. This is why we highlight it separately in thick dotts — its static type would be rcuFresh. Note that we cannot duplicate a rcuFresh variable, nor read a field of an object it points to. This restriction localizes our reasoning about the effects of replacing with a fresh node to just one fresh reference and the object it points to. Otherwise another mechanism would be required to ensure that once a fresh reference was linked into the heap, there were no aliases still typed as fresh — since that would have risked linking the same reference into the heap in two locations.

The transition from the Figure 7a to 7b illustrates the effects of the heap mutation (replacing with a fresh node). The reasoning in the type system for replacing with a fresh node is nearly the same as for unlinking an existing node, with one exception. In replacing with a fresh node, there is no need to consider the paths of nodes deeper in the tree than the point of mutation. In the unlinking case, those nodes' static paths would become invalid. In the case of replacing with a fresh node, those descendants' paths are preserved. Our type rule for ensuring safe replacement (T-REPLACE) prevents path aliasing (representing the nonexistence of a_0 and a_1 via dashed lines and circles) by negating a MayAlias query and prevents field mapping aliasing (nonexistence of any object field from any other context pointing to cr) via asserting $(y \neq o)$. It is important to note that objects(H_4, H_2) in the field mappings of the cr whose referent is to be unlinked captured by the heap node's field mappings referenced by cf in rcuFresh. This is part of enforcing locality on the heap mutation and captured by assertion $\mathcal{N} = \mathcal{N}'$ in the type rule(T-REPLACE).

Inserting a Fresh Node T-INSERT type checks heap updates that link a fresh node into a linked data structure. Inserting a rcuFresh reference also faces some of the aliasing complications that we have already discussed for direct unlinking and replacing a node. Unlike the replacement case, the path to the last heap node (the referent of o) from the root is *extended* by f, which risks falsifying the paths for aliases and descendants of o. The final assumption(the implication) of T-INSERT checks for this inconsistency.

There is also another rule, T-LINKF-NULL, not shown in Figure 6, which handles the case where the fields of the fresh node are not object references, but instead all contain null (e.g., for appending to the end of a linked list or inserting a leaf node in a tree).

Critical Sections (***Referencing inside RCU Blocks***) We introduce the *syntactic sugaring* RCUWrite $x.f$ as y in $\{C\}$ for write-side critical sections where the analogous syntactic sugaring can be found for read-side critical sections in Appendix E of the technical report [21].

The type system ensures unlinked and freeable references are handled linearly, as they cannot be dropped – coerced to undef. The top-level rule TORCUWRITE in Figure 6 ensures unlinked references have been freed by forbidding them in the critical section's post-type environment. Our technical report [21] also includes the analogous rule TORCUREAD for the read critical section in Figure 33 of Appendix E.

Preventing the reuse of rcuItr references across critical sections is subtler: the non-critical section system is not flow-sensitive, and does not include rcuItr. Therefore, the initial environment lacks rcuItr references, and trailing rcuItr references may not escape.

5 Evaluation

We have used our type system to check correct use of RCU primitives in two RCU data structures representative of the broader space.

Figure 1 gives the type-annotated code for `add` and `remove` operations on a linked list implementation of a bag data structure, following McKenney's example [26]. Our technical report [21] contains code for membership checking.

We have also type checked the most challenging part of an RCU binary search tree, the deletion (which also contains the code for a lookup). Our implementation is a slightly simplified version of the Citrus BST [3]: their code supports fine-grained locking for multiple writers, while ours supports only one writer by virtue of using our single-writer primitives. For lack of space the annotated code is only in Appendix B of the technical report [21], but here we emphasise the important aspects our type system via showing its capabilities of typing BST delete method, which also includes looking up for the node to be deleted.

In Figure 8, we show the steps for deleting the heap node H_1. To locate the node H_1, as shown in Figure 8a, we first traverse the subtree T_0 with references pr and cr, where pr is the parent of cr during traversal:

$$pr : rcuItr(l|r)^k\{l|r \to cr\},\ cr : rcuItr(l|r)^k.(l|r)\{\}$$

Traversal of T_0 is summarized as $(l|k)^k$. The most subtle aspect of the deletion is the final step in the case the node H_1 to remove has both children; as shown in Figure 8b, the code must traverse the subtree T_4 to locate the next element in collection order: the node H_s, the left-most node of H_1's right child (sc) and its parent (lp):

$$lp : (l|r)^k.(l|r).r.(l|r)^m\{l|r \to sc\},\ sc : (l|r)^k.(l|r).r.l.(l)^m.l\{\}$$

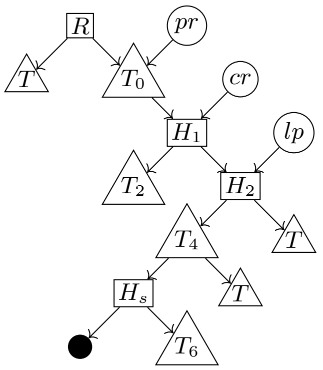
where the traversal of T_4 is summarized as $(l|m)^m$.

Then H_s is copied into a new *freshly-allocated* node as shown in Figure 8b, which is then used to *replace* node H_1 as shown in Figure 8c: the replacement's fields exactly match H_1's except for the data (T-Replace via $\mathcal{N}_1 = \mathcal{N}_2$) as shown in Figure 8b, and the parent is updated to reference the replacement, unlinking H_1.

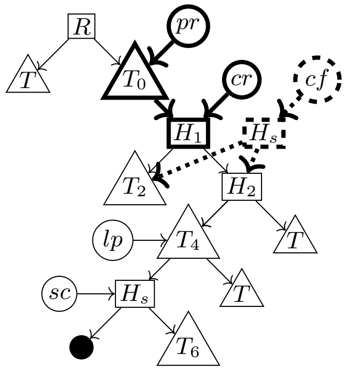
At this point, as shown in Figures 8c and 8d, there are two nodes with the same value in the tree (the *weak* BST property of the Citrus BST [3]): the replacement node, and what was the left-most node under H_1's right child. This latter (original) node H_s must be unlinked as shown in Figure 8e, which is simpler because by being left-most the left child is null, avoiding another round of replacement (T-UnlinkH via $\forall_{f \in dom(\mathcal{N}_1)}.\ f \neq f_2 \implies (\mathcal{N}_1(f) = \mathsf{null})$.

Traversing T_4 to find successor complicates the reasoning in an interesting way. After the successor node H_s is found in 8b, there are *two* local unlinking operations as shown in Figures 8c and 8e, at different depths of the tree. This is why the type system must keep separate abstract iteration counts, e.g., k of $(l|r)^k$ or m of $(l|r)^m$, for traversals in loops — these indices act like multiple cursors into the data structure, and allow the types to carry enough information to keep those changes separate and ensure neither introduces a cycle.

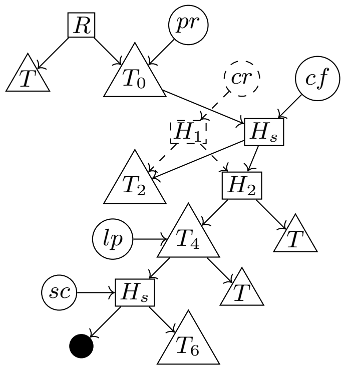
To the best of our knowledge, we are the first to check such code for memory-safe use of RCU primitives modularly, without appeal to the specific implementation of RCU primitives.



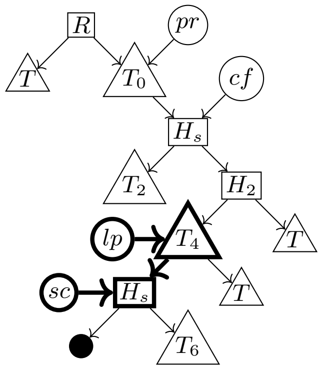
(a) The writer traverses subtree T_0 to find the heap node H_1 with local references pr and cr. Black-filled node representing the null node.



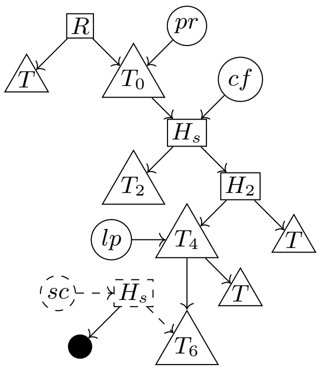
(b) Traverse subtree T_4 starting from H_2 with references lp and sc to find successor H_s of H_1. Duplicating H_s as a fresh heap node before replacing H_1 with the fresh one.



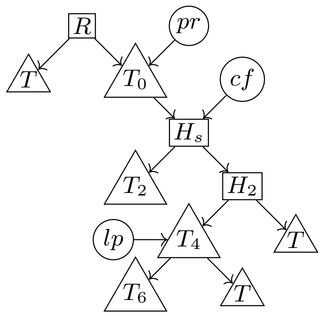
(c) Replace H_1 with fresh successor and synchronize with the readers.



(d) Unlinks old successor referenced by sc.



(e) Safe unlinking of the old successor whose left subtree is null.



(f) Reclamation of the old successor.

Fig. 8: Delete of a heap node with two children in BST [3].

6 Soundness

This section outlines the proof of type soundness – our full proof appears the accompanying technical report [21]. We prove type soundness by embedding the type system into an abstract concurrent separation logic called the Views Framework [9], which when given certain information about proofs for a specific language (primitives and primitive typing) gives back a full program logic including choice and iteration. As with other work taking this approach [15,14], this consists of several key steps explained in the following subsections, but a high-level informal soundness argument is twofold. First, because the parameters given to the Views framework ensure the Views logic's Hoare triples $\{-\}C\{-\}$ are sound, this proves soundness of the type rules with respect to type denotations. Second, as our denotation of types encodes the property that the post-environment of any type rule accurately characterizes which memory is linked vs. unlinked, etc., and the global invariants ensure all allocated heap memory is reachable from the root or from some thread's stack, this entails that our type system prevents memory leaks.

6.1 Proof

This section provides more details on how the Views Framework [9] is used to prove soundness, giving the major parameters to the framework and outlining global invariants and key lemmas.

Logical State Section 3 defined what Views calls *atomic actions* (the primitive operations) and their semantics on runtime *machine states*. The Views Framework uses a separate notion of instrumented (logical) state over which the logic is built, related by a concretization function $\lfloor - \rfloor$ taking an instrumented state to the machine states of Section 3. Most often — including in our proof — the logical state adds useful auxiliary state to the machine state, and the concretization is simply projection. Thus we define our logical states LState as:

- A machine state, $\sigma = (s, h, l, rt, R, B)$
- An observation map, O, of type $\mathsf{Loc} \rightarrow \mathcal{P}(\mathsf{obs})$
- Undefined variable map, U, of type $\mathcal{P}(\mathsf{Var} \times \mathsf{TID})$
- Set of threads, T, of type $\mathcal{P}(\mathsf{TIDS})$
- A to-free map (or free list), F, of type $\mathsf{Loc} \rightharpoonup \mathcal{P}(\mathsf{TID})$

The thread ID set T includes the thread ID of all running threads. The free map F tracks which reader threads may hold references to each location. It is not required for execution of code, and for validating an implementation could be ignored, but we use it later with our type system to help prove that memory deallocation is safe. The (per-thread) variables in the undefined variable map U are those that should not be accessed (e.g., dangling pointers).

The remaining component, the observation map O, requires some further explanation. Each memory allocation / object can be *observed* in one of the following states by a variety of threads, depending on how it was used.

$$\mathsf{obs} := \mathtt{iterator}\ \mathrm{tid} \mid \mathtt{unlinked} \mid \mathtt{fresh} \mid \mathtt{freeable} \mid \mathtt{root}$$

An object can be observed as part of the structure (`iterator`), removed but possibly accessible to other threads, freshly allocated, safe to deallocate, or the root of the structure.

Invariants of RCU Views and Denotations of Types Next, we aim to convey the intuition behind the predicate WellFormed which enforces global invariants on logical states, and how it interacts with the denotations of types (Figure 9) in key ways.

WellFormed is the conjunction of a number of more specific invariants, which we outline here. For full details, see Appendix A.2 of the technical report [21].

The Invariant for Read Traversal Reader threads access valid heap locations even during the grace period. The validity of their heap accesses ensured by the observations they make over the heap locations — which can only be iterator as they can only use local rcuItr references. To this end, a Readers-Iterators-Only invariant asserts that reader threads can only observe a heap location as iterator.

Invariants on Grace-Period Our logical state includes a "free list" auxiliary state tracking which readers are still accessing *each* unlinked node during grace periods. This must be consistent with the bounding thread set B in the machine state, and this consistency is asserted by the Readers-In-Free-List invariant. This is essentially tracking which readers are being "shown grace" for each location. The Iterators-Free-List invariant complements this by asserting all readers with such observations on unlinked nodes are in the bounding thread set.

The writer thread can refer to a heap location in the free list with a local reference either in type freeable or unlinked. Once the writer unlinks a heap node, it first observes the heap node as unlinked then freeable. The denotation of freeable is only valid following a grace period: it asserts no readers hold aliases of the freeable reference. The denotation of unlinked permits the either the same (perhaps no readers overlapped) or that it is in the to-free list.

Invariants on Safe Traversal against Unlinking The write-side critical section must guarantee that no updates to the heap cause invalid memory accesses. The Writer-Unlink invariant asserts that a heap location observed as iterator by the writer thread cannot be observed differently by other threads. The denotation of the writer thread's rcuItr reference, $[\![\mathsf{rcuItr}\,\rho\,\mathcal{N}]\!]_{tid}$, asserts that following a path from the root compatible with ρ reaches the referent, and all are observed as iterator.

The denotation of a reader thread's rcuItr reference, $[\![\mathsf{rcuItr}]\!]_{tid}$ and the invariants Readers-Iterator-Only, Iterators-Free-List and Readers-In-Free-List all together assert that a reader thread(which can also be a bounding thread) can view an unlinked heap location(which can be in the free list) only as iterator. At the same time, it is essential that reader threads arriving after a node is unlinked cannot access it. The invariants Unlinked-Reachability and Free-List-Reachability ensure that any unlinked nodes are reachable only from other unlinked nodes, and never from the root.

$$
\begin{array}{lcl}
[\![x : \mathsf{rcultr}\,\rho\,\mathcal{N}]\!]_{tid} & = & \left\{ m \in \mathcal{M} \,\middle|\, \begin{array}{l} (\mathsf{iterator}\,tid \in O(s(x,tid))) \wedge (x \notin U) \\ \wedge(\forall_{f_i \in dom(\mathcal{N}) x_i \in codom(\mathcal{N})} . \left\{ \begin{array}{l} s(x_i,tid) = h(s(x,tid),f_i) \\ \wedge \mathsf{iterator} \in O(s(x_i,tid))) \end{array} \right. \\ \wedge(\forall_{\rho',\rho''} . \rho'.\rho'' = \rho \implies \mathsf{iterator}\,tid \in O(h^*(rt,\rho'))) \\ \wedge h^*(rt,\rho) = s(x,tid) \wedge (l = tid \wedge s(x,\_) \notin dom(F))) \end{array} \right\} \\
[\![x : \mathsf{rcultr}]\!]_{tid} & = & \left\{ m \in \mathcal{M} \,\middle|\, \begin{array}{l} (\mathsf{iterator}\,tid \in O(s(x,tid))) \wedge (x \notin U) \wedge \\ (tid \in B) \implies \left\{ \begin{array}{l} (\exists_{T' \subseteq B} . \{s(x,tid) \mapsto T'\} \cap F \neq \emptyset) \wedge \\ \wedge(tid \in T') \end{array} \right. \end{array} \right\} \\
[\![x : \mathsf{unlinked}]\!]_{tid} & = & \left\{ m \in \mathcal{M} \,\middle|\, \begin{array}{l} (\mathsf{unlinked} \in O(.s(x,tid)) \wedge l = tid \wedge x \notin U) \wedge \\ (\exists_{T' \subseteq T} . s(x,tid) \mapsto T' \in F \implies T' \subseteq B \wedge tid \notin T') \end{array} \right\} \\
[\![x : \mathsf{freeable}]\!]_{tid} & = & \left\{ m \in \mathcal{M} \,\middle|\, \begin{array}{l} \mathsf{freeable} \in O(s(x,tid)) \wedge l = tid \wedge x \notin U \wedge \\ s(x,tid) \mapsto \{\emptyset\} \in F \end{array} \right\} \\
[\![x : \mathsf{rcuFresh}\,\mathcal{N}]\!]_{tid} & = & \left\{ m \in \mathcal{M} \,\middle|\, \begin{array}{l} (\mathsf{fresh} \in O(s(x,tid)) \wedge x \notin U \wedge s(x,tid) \notin dom(F)) \\ (\forall_{f_i \in dom(\mathcal{N}), x_i \in codom(\mathcal{N})} . s(x_i,tid) = h(s(x,tid),f_i) \\ \wedge \mathsf{iterator}\,tid \in O(s(x_i,tid)) \wedge s(x_i,tid) \notin dom(F)) \end{array} \right\} \\
[\![x : \mathsf{undef}]\!]_{tid} & = & \left\{ m \in \mathcal{M} \,\middle|\, (x,tid) \in U \wedge s(x,tid) \notin dom(F) \right\} \\
[\![x : \mathsf{rcuRoot}]\!]_{tid} & = & \left\{ m \in \mathcal{M} \,\middle|\, \begin{array}{l} ((rt \notin U \wedge s(x,tid) = rt \wedge rt \in dom(h) \wedge \\ O(rt) \in \mathsf{root} \wedge s(x,tid) \notin dom(F)) \end{array} \right\}
\end{array}
$$

provided $h^* : (\mathsf{Loc} \times \mathsf{Path}) \rightharpoonup \mathsf{Val}$

Fig. 9: Type Environments

Invariants on Safe Traversal against Inserting/Replacing A writer replacing an existing node with a fresh one or inserting a single fresh node assumes the fresh (before insertion) node is unreachable to readers before it is published/linked. The Fresh-Writes invariant asserts that a fresh heap location can only be allocated and referenced by the writer thread. The relation between a freshly allocated heap and the rest of the heap is established by the Fresh-Reachable invariant, which requires that there exists no heap node pointing to the freshly allocated one. This invariant supports the preservation of the tree structure. The Fresh-Not-Reader invariant supports the safe traversal of the reader threads via asserting that they cannot observe a heap location as fresh. Moreover, the denotation of the rcuFresh type, $[\![\mathsf{rcuFresh}\,\mathcal{N}]\!]_{tid}$, enforces that fields in $\mathcal{N}$ point to valid heap locations (observed as iterator by the writer thread).

Invariants on Tree Structure Our invariants enforce the *tree* structure heap layouts for data structures. The Unique-Reachable invariant asserts that every heap location reachable from root can only be reached with following an unique path. To preserve the tree structure, Unique-Root enforces unreachability of the root from any heap location that is reachable from root itself.

Type Environments Assertions in the Views logic are (almost) sets of the logical states that satisfy a validity predicate WellFormed, outlined above:

$$\mathcal{M} \overset{def}{=} \{m \in (\mathsf{MState} \times O \times U \times T \times F) \mid \mathsf{WellFormed}(m)\}$$

Every type environment represents a set of possible views (WellFormed logical states) consistent with the types in the environment. We make this precise with a denotation function

$$[\![-]\!]_{\_} : \mathsf{TypeEnv} \to \mathsf{TID} \to \mathcal{P}(\mathcal{M})$$

$$\bullet \overset{\text{def}}{=} (\bullet_\sigma, \bullet_O, \cup, \cup) \quad (F_1 \bullet_F F_2) \overset{\text{def}}{=} F_1 \cup F_2 \ \texttt{when}\ dom(F_1) \cap dom(F_2) = \emptyset$$

$$O_1 \bullet_O O_2(loc) \overset{\text{def}}{=} O_1(loc) \cup O_2(loc) \quad (s_1 \bullet_s s_2) \overset{\text{def}}{=} s_1 \cup s_2 \ \texttt{when}\ dom(s_1) \cap dom(s_2) = \emptyset$$

$$(h_1 \bullet_h h_2)(o,f) \overset{\text{def}}{=} \begin{cases} \text{undef} & \text{if } h_1(o,f) = v \wedge h_2(o,f) = v' \wedge v' \neq v \\ v & \text{if } h_1(o,f) = v \wedge h_2(o,f) = v \\ v & \text{if } h_1(o,f) = \text{undef} \wedge h_2(o,f) = v \\ v & \text{if } h_1(o,f) = v \wedge h_2(o,f) = \text{undef} \\ \text{undef} & \text{if } h_1(o,f) = \text{undef} \wedge h_2(o,f) = \text{undef} \end{cases}$$

$$((s,h,l,rt,R,B),O,U,T,F)\mathcal{R}_0((s',h',l',rt',R',B'),O',U',T',F') \overset{\text{def}}{=}$$
$$\bigwedge \left\{ \begin{array}{l} l \in T \to (h = h' \wedge l = l') \\ l \in T \to F = F' \\ \forall tid, o.\ \mathsf{iterator}\ tid \in O(o) \to o \in dom(h) \\ \forall tid, o.\ \mathsf{iterator}\ tid \in O(o) \to o \in dom(h') \\ \forall tid, o.\ \mathsf{root}\ tid \in O(o) \to o \in dom(h) \\ \forall tid, o.\ \mathsf{root}\ tid \in O(o) \to o \in dom(h') \\ O = O' \wedge U = U' \wedge T = T' \wedge R = R' \wedge rt = rt' \\ \forall x, t \in T.\ s(x,t) = s'(x,t) \end{array} \right\}$$

Fig. 10: Composition($\bullet$) and Thread Interference Relation($\mathcal{R}_0$)

that yields the set of states corresponding to a given type environment. This is defined as the intersection of individual variables' types as in Figure 9.

Individual variables' denotations are extended to context denotations slightly differently depending on whether the environment is a reader or writer thread context: writer threads own the global lock, while readers do not:

- For read-side as $[\![x_1 : T_1, \ldots x_n : T_n]\!]_{tid,\mathsf{R}} = [\![x_1 : T_1]\!]_{tid} \cap \ldots \cap [\![x_n : T_n]\!]_{tid} \cap [\![\mathsf{R}]\!]_{tid}$ where $[\![\mathsf{R}]\!]_{tid} = \{(s,h,l,rt,R,B),O,U,T,F \mid tid \in R\}$
- For write-side as $[\![x_1 : T_1, \ldots x_n : T_n]\!]_{tid,\mathsf{M}} = [\![x_1 : T_1]\!]_{tid} \cap \ldots \cap [\![x_n : T_n]\!]_{tid} \cap [\![\mathsf{M}]\!]_{tid}$ where $[\![\mathsf{M}]\!]_{tid} = \{(s,h,l,rt,R,B),O,U,T,F \mid tid = l\}$

Composition and Interference To support framing (weakening), the Views Framework requires that views form a partial commutative monoid under an operation $\bullet : \mathcal{M} \longrightarrow \mathcal{M} \longrightarrow \mathcal{M}$, provided as a parameter to the framework. The framework also requires an interference relation $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$ between views to reason about local updates to one view preserving validity of adjacent views (akin to the small-footprint property of separation logic). Figure 10 defines our composition operator and the core interference relation $\mathcal{R}_0$ — the actual inferference between views (between threads, or between a local action and framed-away state) is the reflexive transitive closure of $\mathcal{R}_0$. Composition is mostly straightforward point-wise union (threads' views may overlap) of each component. Interference bounds the interference writers and readers may inflict on each other. Notably, if a view contains the writer thread, other threads may not modify the shared portion of the heap, or release the writer lock. Other aspects of interference are natural restrictions like that threads may not modify each others' local variables. WellFormed states are closed under both composition (with another WellFormed state) and interference ($\mathcal{R}$ relates WellFormed states only to other WellFormed states).

Stable Environment and Views Shift The framing/weakening type rule will be translated to a use of the frame rule in the Views Framework's logic. There

$$\downarrow \mathsf{if}\ (x.f == y)\ C_1\ C_2 \downarrow tid \overset{\text{def}}{=} z = x.f; ((\mathsf{assume}(z = y); C_1) + (\mathsf{assume}(z \neq y); C_2))$$

$$[\![\mathtt{assume}(\mathcal{S})]\!](s) \overset{\text{def}}{=} \begin{cases} \{s\} & \text{if } s \in \mathcal{S} \\ \emptyset & \text{Otherwise} \end{cases} \quad \downarrow \mathsf{while}\ (e)\ C \downarrow \overset{\text{def}}{=} (\mathsf{assume}(e); C)^*\ ; (\mathsf{assume}(\neg e));$$

$$\frac{\{P\} \cap \{\lceil \mathcal{S} \rceil\} \sqsubseteq \{Q\}}{\{P\}\mathtt{assume}\,(\mathcal{S})\,\{Q\}} \quad \text{where} \quad \lceil \mathcal{S} \rceil = \{m | \lfloor m \rfloor \cap \mathcal{S} \neq \emptyset\}$$

Fig. 11: Encoding branch conditions with assume(b)

separating conjunction is simply the existence of two composable instrumented states:

$$m \in P * Q \overset{def}{=} \exists m'.\, \exists m''.\, m' \in P \wedge m'' \in Q \wedge m \in m' \bullet m''$$

In order to validate the frame rule in the Views Framework's logic, the assertions in its logic — sets of well-formed instrumented states — must be restricted to sets of logical states that are *stable* with respect to expected interference from other threads or contexts, and interference must be compatible in some way with separating conjunction. Thus a View — the actual base assertions in the Views logic — are then:

$$\mathsf{View}_{\mathcal{M}} \overset{def}{=} \{M \in \mathcal{P}(\mathcal{M}) | \mathcal{R}(M) \subseteq M\}$$

Additionally, interference must distribute over composition:

$$\forall m_1, m_2, m.\, (m_1 \bullet m_2)\mathcal{R}m \implies \exists m_1' m_2'.\, m_1 \mathcal{R} m_1' \wedge m_2 \mathcal{R} m_2' \wedge m \in m_1' \bullet m_2'$$

Because we use this induced Views logic to prove soundness of our type system by translation, we must ensure any type environment denotes a valid view:

Lemma 1 (Stable Environment Denotation-M). *For any* closed *environment Γ (i.e., $\forall x \in \mathsf{dom}(\Gamma).,\mathsf{FV}(\Gamma(x)) \subseteq \mathsf{dom}(\Gamma)$): $\mathcal{R}([\![\Gamma]\!]_{\mathsf{M},tid}) \subseteq [\![\Gamma]\!]_{\mathsf{M},tid}$. Alternatively, we say that environment denotation is* stable *(closed under $\mathcal{R}$).*

Proof. In Appendix A.1 Lemma 7 of the technical report [21].

We elide the statement of the analogous result for the read-side critical section, available in Appendix A.1 of the technical report.

With this setup done, we we can state the connection between the Views Framework logic induced by earlier parameters, and the type system from Section 4. The induced Views logic has a familiar notion of Hoare triple — $\{p\}C\{q\}$ where p and q are elements of $\mathsf{View}_{\mathcal{M}}$ — with the usual rules for non-deterministic choice, non-deterministic iteration, sequential composition, and parallel composition, sound given the proof obligations just described above. It is parameterized by a rule for atomic commands that requires a specification of the triples for primitive operations, and their soundness (an obligation we must prove). This can then be used to prove that every typing derivation embeds to a valid derivation in the Views Logic, roughly $\forall \Gamma, C, \Gamma', tid.\, \Gamma \vdash C \dashv \Gamma' \Rightarrow \{[\![\Gamma]\!]_{tid}\}[\![C]\!]_{tid}\{[\![\Gamma']\!]_{tid}\}$ once for the writer type system, once for the readers.

There are two remaining subtleties to address. First, commands C also require translation: the Views Framework has only non-deterministic branches and loops, so the standard versions from our core language must be encoded. The approach to this is based on a standard idea in verification, which we show here for conditionals as shown in Figure 11. $\mathsf{assume}(b)$ is a standard idea in verification semantics [4,31], which "does nothing" (freezes) if the condition b is false, so its postcondition in the Views logic can reflect the truth of b. assume in Figure 11 adapts this for the Views Framework as in other Views-based proofs [15,14], specifying sets of machine states as a predicate. We write boolean expressions as shorthand for the set of machine states making that expression true. With this setup done, the top-level soundness claim then requires proving – once for the reader type system, once for the writer type system – that every valid source typing derivation corresponds to a valid derivation in the Views logic: $\forall \Gamma, C, \Gamma', \Gamma \vdash_M C \dashv \Gamma' \Rightarrow \{\llbracket\Gamma\rrbracket\} \downarrow C \downarrow \{\llbracket\Gamma'\rrbracket\}$.

Second, we have not addressed a way to encode subtyping. One might hope this corresponds to a kind of implication, and therefore subtyping corresponds to consequence. Indeed, this is how we (and prior work [15,14]) address subtyping in a Views-based proof. Views defines the notion of *view shift*[2] ($\sqsubseteq$) as a way to reinterpret a set of instrumented states as a new (compatible) set of instrumented states, offering a kind of logical consequence, used in a rule of consequence in the Views logic:

$$p \sqsubseteq q \overset{def}{=} \forall m \in \mathcal{M}.\ \lfloor p * \{m\} \rfloor \subseteq \lfloor q * \mathcal{R}(\{m\}) \rfloor$$

We are now finally ready to prove the key lemmas of the soundness proof, relating subtying to view shifts, proving soundness of the primitive actions, and finally for the full type system. These proofs occur once for the writer type system, and once for the reader; we show here only the (more complex) writer obligations:

Lemma 2 (Axiom of Soundness for Atomic Commands). *For each axiom,* $\Gamma_1 \vdash_M \alpha \dashv \Gamma_2$, *we show* $\forall m.\ \llbracket\alpha\rrbracket(\lfloor \llbracket\Gamma_1\rrbracket_{tid} * \{m\} \rfloor) \subseteq \lfloor \llbracket\Gamma_2\rrbracket_{tid} * \mathcal{R}(\{m\}) \rfloor$

Proof. By case analysis on α. Details in Appendix A.1 of the techical report [21].

Lemma 3 (Context-SubTyping-M). $\Gamma \prec: \Gamma' \implies \llbracket\Gamma\rrbracket_{M,tid} \sqsubseteq \llbracket\Gamma'\rrbracket_{M,tid}$

Proof. Induction on the subtyping derivation, then inducting on the single-type subtype relation for the first variable in the non-empty context case.

Lemma 4 (Views Embedding for Write-Side).
$\forall \Gamma, C, \Gamma', t.\ \Gamma \vdash_M C \dashv \Gamma' \Rightarrow \llbracket\Gamma\rrbracket_t \cap \llbracket M\rrbracket_t \vdash \llbracket C\rrbracket_t \dashv \llbracket\Gamma'\rrbracket_t \cap \llbracket M\rrbracket_t$

Proof. By induction on the typing derivation, appealing to Lemma 2 for primitives, Lemma 3 and consequence for subtyping, and otherwise appealing to structural rules of the Views logic and inductive hypotheses. Full details in Appendix A.1 of the technical report [21].

[2] This is the same notion present in later program logics like Iris [19], though more recent variants are more powerful.

The corresponding obligations and proofs for the read-side critical section type system are similar in statement and proof approach, just for the read-side type judgments and environment denotations.

7 Discussion & Related Work

Our type system builds on a great deal of related work on RCU implementations and models; and general concurrent program verification. Due to space limit, this section captures only discussions on program logics, modeling RCU and memory models, but our technical report [21] includes detailed discussions on model-checking [18,22,8], language oriented approaches [17,6,17] and realization of our semantics in an implementation as well.

Modeling RCU and Memory Models Alglave et al. [2] propose a memory model to be assumed by the platform-independent parts of the Linux kernel, regardless of the underlying hardware's memory model. As part of this, they give the first formalization of what it means for an RCU implementation to be correct (previously this was difficult to state, as the guarantees in principle could vary by underlying CPU architecture). Essentially, reader critical sections must not span grace periods. They prove by hand that the Linux kernel RCU implementation [1] satisfies this property. McKenney has defined fundamental requirements of RCU implementations [27]; our model in Section 3 is a valid RCU implementation according to those requirements (assuming sequential consistency) aside from one performance optimization, *Read-to-Write Upgrade*, which is important in practice but not memory-safety centric – see the technical report [21] for detailed discussion on satisfying RCU requirements. To the best of our knowledge, ours is the first abstract *operational* model for a Linux kernel-style RCU implementation – others are implementation-specific [23] or axiomatic like Alglave et al.'s.

Tassarotti et al. model a well-known way of implementing RCU synchronization without hurting readers' performance — Quiescent State Based Reclamation (QSBR) [8] — where synchronization between the writer thread and reader threads occurs via per-thread counters. Tassarotti et al. [33] uses a protocol based program logic based on separation and ghost variables called GPS [35] to verify a user-level implementation of RCU with a singly linked list client under *release-acquire* semantics, which is a weaker memory model than sequential-consistency. Despite the weaker model, the protocol that they enforce on their RCU primitives is nearly the same what our type system requires. The reads and writes to per thread QSBR structures are similar to our more abstract updates to reader and bounding sets. Therefore, we anticipate it would be possible to extend our type system in the future for similar weak memory models.

Program Logics Fu et al. [13] extend Rely-Guarantee and Separation-Logic [36,11,10] with the *past-tense* temporal operator to eliminate the need for using a history variable and lift the standard separation conjunction to assert over on execution histories. Gotsman et al. [16] take assertions from temporal logic to separation logic [36] to capture the essence of epoch-based memory reclamation algorithms and have a simpler proof than what Fu et al. have [13] for Michael's non-blocking stack [30] implementation under a sequentially consistent memory model.

Tassarotti et al. [33] use *abstract-predicates* – e.g. WriterSafe – that are specialized to the singly-linked structure in their evaluation. This means reusing their ideas for another structure, such as a binary search tree, would require revising many of their invariants. By contrast, our types carry similar information (our denotations are similar to their definitions), but are reusable across at least singly-linked and tree data structures (Section 5). Their proofs of a linked list also require managing assertions about RCU implementation resources, while these are effectively hidden in the type denotations in our system. On the other hand, their proofs ensure full functional correctness. Meyer and Wolff [29] make a compelling argument that separating memory safety from correctness if profitable, and we provide such a decoupled memory safety argument.

8 Conclusions

We presented the first type system that ensures code uses RCU memory management safely, and which is significantly simpler than full-blown verification logics. To this end, we gave the first general operational model for RCU-based memory management. Based on our suitable abstractions for RCU in the operational semantics we are the first showing that decoupling the *memory-safety* proofs of RCU clients from the underlying reclamation model is possible. Meyer et al. [29] took similar approach for decoupling the *correctness* verification of the data structures from the underlying reclamation model under the assumption of the *memory-safety* for the data structures. We demonstrated the applicability/reusability of our types on two examples: a linked-list based bag [26] and a binary search tree [3]. To our best knowledge, we are the first presenting the *memory-safety* proof for a tree client of RCU. We managed to prove type soundness by embedding the type system into an abstract concurrent separation logic called the Views Framework [9] and encode many RCU properties as either type-denotations or global invariants over abstract RCU state. By doing this, we managed to discharge these invariants once as a part of soundness proof and did not need to prove them for each different client.

Acknowledgements. We are grateful to Matthew Parkinson for guidance and productive discussions on the early phase of this project. We also thank to Nik Sultana and Klaus V. Gleissenthall for their helpful comments and suggestions for improving the paper.

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. pp. 141–157 (2013). https://doi.org/10.1007/978-3-642-39799-8_9, `http://dx.doi.org/10.1007/978-3-642-39799-8_9`

2. Alglave, J., Maranget, L., McKenney, P.E., Parri, A., Stern, A.: Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 405–418. ASPLOS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3173162.3177156, `http://doi.acm.org/10.1145/3173162.3177156`
3. Arbel, M., Attiya, H.: Concurrent updates with rcu: Search tree as an example. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing. pp. 196–205. PODC '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2611462.2611471, `http://doi.acm.org/10.1145/2611462.2611471`
4. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Proceedings of the 4th International Conference on Formal Methods for Components and Objects. pp. 364–387. FMCO'05, Springer-Verlag, Berlin, Heidelberg (2006). https://doi.org/10.1007/11804192_17, `http://dx.doi.org/10.1007/11804192_17`
5. Clements, A.T., Kaashoek, M.F., Zeldovich, N.: Scalable address spaces using RCU balanced trees. In: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012. pp. 199–210 (2012). https://doi.org/10.1145/2150976.2150998, `http://doi.acm.org/10.1145/2150976.2150998`
6. Cooper, T., Walpole, J.: Relativistic programming in haskell using types to enforce a critical section discipline (2015), `http://web.cecs.pdx.edu/~walpole/papers/haskell2015.pdf`
7. Desnoyers, M., McKenney, P.E., Dagenais, M.R.: Multi-core systems modeling for formal verification of parallel algorithms. SIGOPS Oper. Syst. Rev. **47**(2), 51–65 (Jul 2013). https://doi.org/10.1145/2506164.2506174, `http://doi.acm.org/10.1145/2506164.2506174`
8. Desnoyers, M., McKenney, P.E., Stern, A., Walpole, J.: User-level implementations of read-copy update. IEEE Transactions on Parallel and Distributed Systems (2009), `/static/publications/desnoyers-ieee-urcu-submitted.pdf`
9. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: compositional reasoning for concurrent programs. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 287–300 (2013). https://doi.org/10.1145/2429069.2429104, `http://doi.acm.org/10.1145/2429069.2429104`
10. Feng, X.: Local rely-guarantee reasoning. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 315–327. POPL '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1480881.1480922, `http://doi.acm.org/10.1145/1480881.1480922`
11. Feng, X., Ferreira, R., Shao, Z.: On the relationship between concurrent separation logic and assume-guarantee reasoning. In: Proceedings of the 16th European Symposium on Programming. pp. 173–188. ESOP'07, Springer-Verlag, Berlin, Heidelberg (2007), `http://dl.acm.org/citation.cfm?id=1762174.1762193`
12. Fu, M., Li, Y., Feng, X., Shao, Z., Zhang, Y.: Reasoning about optimistic concurrency using a program logic for history. In: CONCUR. pp. 388–402. Springer (2010)

13. Fu, M., Li, Y., Feng, X., Shao, Z., Zhang, Y.: Reasoning about optimistic concurrency using a program logic for history. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory. pp. 388–402. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
14. Gordon, C.S., Ernst, M.D., Grossman, D., Parkinson, M.J.: Verifying Invariants of Lock-free Data Structures with Rely-Guarantee and Refinement Types. ACM Transactions on Programming Languages and Systems (TOPLAS) **39**(3) (May 2017). https://doi.org/10.1145/3064850, `http://doi.acm.org/10.1145/3064850`
15. Gordon, C.S., Parkinson, M.J., Parsons, J., Bromfield, A., Duffy, J.: Uniqueness and Reference Immutability for Safe Parallelism. In: Proceedings of the 2012 ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'12). Tucson, AZ, USA (October 2012). https://doi.org/10.1145/2384616.2384619, `http://dl.acm.org/citation.cfm?id=2384619`
16. Gotsman, A., Rinetzky, N., Yang, H.: Verifying concurrent memory reclamation algorithms with grace. In: Proceedings of the 22Nd European Conference on Programming Languages and Systems. pp. 249–269. ESOP'13, Springer-Verlag, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_15, `http://dx.doi.org/10.1007/978-3-642-37036-6_15`
17. Howard, P.W., Walpole, J.: A relativistic enhancement to software transactional memory. In: Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism. pp. 15–15. HotPar'11, USENIX Association, Berkeley, CA, USA (2011), `http://dl.acm.org/citation.cfm?id=2001252.2001267`
18. Kokologiannakis, M., Sagonas, K.: Stateless model checking of the linux kernel's hierarchical read-copy-update (tree rcu). In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. pp. 172–181. SPIN 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3092282.3092287, `http://doi.acm.org/10.1145/3092282.3092287`
19. Krebbers, R., Jung, R., Bizjak, A., Jourdan, J.H., Dreyer, D., Birkedal, L.: The essence of higher-order concurrent separation logic. In: European Symposium on Programming. pp. 696–723. Springer (2017)
20. Kung, H.T., Lehman, P.L.: Concurrent manipulation of binary search trees. ACM Trans. Database Syst. **5**(3), 354–382 (Sep 1980). https://doi.org/10.1145/320613.320619, `http://doi.acm.org/10.1145/320613.320619`
21. Kuru, I., Gordon, C.S.: Safe deferred memory reclamation with types. CoRR **abs/1811.11853** (2018), `http://arxiv.org/abs/1811.11853`
22. Liang, L., McKenney, P.E., Kroening, D., Melham, T.: Verification of the tree-based hierarchical read-copy update in the linux kernel. CoRR **abs/1610.03052** (2016), `http://arxiv.org/abs/1610.03052`
23. Mandrykin, M.U., Khoroshilov, A.V.: Towards deductive verification of c programs with shared data. Program. Comput. Softw. **42**(5), 324–332 (Sep 2016). https://doi.org/10.1134/S0361768816050054, `http://dx.doi.org/10.1134/S0361768816050054`
24. Mckenney, P.E.: Exploiting Deferred Destruction: An Analysis of Read-copy-update Techniques in Operating System Kernels. Ph.D. thesis, Oregon Health & Science University (2004), aAI3139819
25. McKenney, P.E.: N4037: Non-transactional implementation of atomic tree move (May 2014), `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4037.pdf`

26. McKenney, P.E.: Some examples of kernel-hacker informal correctness reasoning. Technical Report paulmck.2015.06.17a (2015), `http://www2.rdrop.com/users/paulmck/techreports/IntroRCU.2015.06.17a.pdf`
27. Mckenney, P.E.: A tour through rcu's requirements (2017), `https://www.kernel.org/doc/Documentation/RCU/Design/Requirements/Requirements.html`
28. Mckenney, P.E., Appavoo, J., Kleen, A., Krieger, O., Krieger, O., Russell, R., Sarma, D., Soni, M.: Read-copy update. In: In Ottawa Linux Symposium. pp. 338–367 (2001)
29. Meyer, R., Wolff, S.: Decoupling lock-free data structures from memory reclamation for static analysis. PACMPL **3**(POPL), 58:1–58:31 (2019), `https://dl.acm.org/citation.cfm?id=3290371`
30. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (Jun 2004). https://doi.org/10.1109/TPDS.2004.8, `http://dx.doi.org/10.1109/TPDS.2004.8`
31. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583. pp. 41–62. VMCAI 2016, Springer-Verlag New York, Inc., New York, NY, USA (2016). https://doi.org/10.1007/978-3-662-49122-5_2, `http://dx.doi.org/10.1007/978-3-662-49122-5_2`
32. Paul E. McKenney, Mathieu Desnoyers, L.J., Triplett, J.: The rcu-barrier menagerie (Nov 2016), `https://lwn.net/Articles/573497/`
33. Tassarotti, J., Dreyer, D., Vafeiadis, V.: Verifying read-copy-update in a logic for weak memory. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 110–120. PLDI '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737992, `http://doi.acm.org/10.1145/2737924.2737992`
34. Triplett, J., McKenney, P.E., Walpole, J.: Resizable, scalable, concurrent hash tables via relativistic programming. In: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference. pp. 11–11. USENIXATC'11, USENIX Association, Berkeley, CA, USA (2011), `http://dl.acm.org/citation.cfm?id=2002181.2002192`
35. Turon, A., Vafeiadis, V., Dreyer, D.: Gps: Navigating weak memory with ghosts, protocols, and separation. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. pp. 691–707. OOPSLA '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2660193.2660243, `http://doi.acm.org/10.1145/2660193.2660243`
36. Vafeiadis, V., Parkinson, M.: A Marriage of Rely/Guarantee and Separation Logic. In: Caires, L., Vasconcelos, V. (eds.) CONCUR 2007 âĂŞ Concurrency Theory, Lecture Notes in Computer Science, vol. 4703, pp. 256–271. Springer Berlin / Heidelberg (2007), `http://dx.doi.org/10.1007/978-3-540-74407-8_18`, 10.1007/978-3-540-74407-8_18