Safe Deferred Memory Reclamation with Types

Ismail Kuru and Colin S. Gordon
Drexel University, Philadelphia PA 19104, USA
ik335@drexel.edu, 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 placed on a deferred free list, 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 [29] 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 have observed a node they wish to deallocate. There are now RCU implementations of many common tree data structures [10, 16, 25, 31, 21], 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, 9, 20, 1], but this does not guarantee correctness of arbitrary client code. Sophisticated verification logics can prove correctness of the RCU primitives and clients [15, 14, 63, 23]. But these techniques require significant verification expertise to apply, and are specialized to individual data structures or implementations. One of the important reasons of the sophistication in the logics stems from the complexity of underlying memory reclamation model. However, Meyer and Wolff [?] 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 (like having a tree memory footprint) common to RCU data structures. 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 significantly 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 many-reader single-writer reader-writer locks. 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 Appendix C. Algorithically, 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 imme-

---

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

---
diately 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 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 [10], 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* [31]: 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* (marking/unlinking) is followed by a *physical deletion* (free). 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 the 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 $\text{Var} \times \text{TID} \rightarrow \text{Loc}$
- A heap, $h$, of type $\text{Loc} \times \text{FName} \rightarrow \text{Val}$
- A lock, $l$, of type $\text{TID} \cup \{\text{unlocked}\}$
- A root location $rt$ of type $\text{Loc}$
- A read set, $R$, of type $\mathcal{P}(\text{TID})$ and
- A bounding set, $B$, of type $\mathcal{P}(\text{TID})$

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, as 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 $\text{WriteBegin}$ and $\text{WriteEnd}$, which acquire and release the lock that enforces mutual exclusion between writers. $\text{WriteBegin}$ only reduces (acquires) if the lock is unlocked.

Standard RCU APIs include a primitive $\text{synchronize_rcu()}$ to wait for a grace period for the current readers. We decompose this here into two actions,
\[
\alpha ::= \text{skip} \mid x.f = y \mid y = x \mid y = x.f \mid y = \text{new} \mid \text{Free}(x) \mid \text{Sync}
\]

\[
\begin{align*}
&\text{[Sync]} \quad \text{id} \quad \triangleq \quad \text{SyncStart; SyncStop} \\
&\text{[RCUWrite} \quad x.f = y \quad \text{as} \quad \text{y in} \quad \{C\}] \quad \text{id} \quad \triangleq \quad \text{WriteBegin;}\{C\}; \text{WriteEnd} \\
&\text{[RCURead} \quad x.f = y \quad \text{as} \quad \text{y in} \quad \{C\}] \quad \text{id} \quad \triangleq \quad \text{ReadBegin;}\{C\}; \text{ReadEnd}
\end{align*}
\]

\begin{itemize}
\item \text{(RCU-WBegin)} \quad \text{[WriteBegin]} \quad (s, h, \text{unlocked}, rt, R, B) \quad \text{id} \quad \triangleq \quad \text{id} \text{[\{s, h, l, rt, R, B\}]
\item \text{(RCU-RBegin)} \quad \text{[ReadBegin]} \quad (s, h, \text{tid}, rt, R) \quad \text{id} \quad \triangleq \quad \text{id} \text{[\{s, h, l, rt, R\}]
\item \text{(RCU-REnd)} \quad \text{[ReadEnd]} \quad (s, h, \text{tid}, rt, R) \quad \text{id} \quad \triangleq \quad \text{id} \text{[\{s, h, l, rt, R\}]
\item \text{(RCU-SSStop)} \quad \text{[SyncStop]} \quad (s, h, l, rt, R, B) \quad \text{id} \quad \triangleq \quad \text{id} \text{[\{s, h, l, rt, R, B\}]
\item \text{[Free]} \quad \text{[Free(x)]} \quad (s, h, l, rt, R, B) \quad \text{id} \quad \triangleq \quad \text{id} \text{[\{s, h, l, rt, R, B\}]
\end{itemize}

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

\[
\begin{align*}
&\text{[HUPDT]} \quad \text{[x.f=x]} \quad (s, h, l, rt, R) \quad \text{id} \quad \triangleq \quad \text{id} \text{[\{s, h, l, rt, R\}], } f \mapsto s(y, tid) \quad l, rt, R, B) \\
&\text{[HREAD]} \quad \text{[y=x.f]} \quad (s, h, l, rt, R) \quad \text{id} \quad \triangleq \quad \text{id} \text{[\{s(x, tid), f\}], } h, l, rt, R, B) \\
&\text{[SUPDT]} \quad \text{[y=\text{new}]} \quad (s, h, l, rt, R) \quad \text{id} \quad \triangleq \quad \text{id} \text{[\{s, h, l, rt, R\}], } x, tid \mapsto (x, tid) \quad l, rt, R, B) \\
&\text{[HALLOC]} \quad \text{[y=\text{null}]} \quad (s, h, l, rt, R) \quad \text{id} \quad \triangleq \quad \text{id} \text{[\{s, h, l, rt, R\}], } x, tid \mapsto (\text{nullmap}, x, tid) \quad l, rt, R, B)
\end{align*}
\]

provided \(rt \neq s(y, tid)\) and \(s(y, tid) \mapsto \ell\), and \(h(\ell \mapsto \text{nullmap}) \triangleq \lambda (o', f), \text{if } o = o' \text{ then skip else } h(o', f)\)

Fig. 2: Operational Semantics for RCU

\textbf{SyncStart} and \textbf{SyncStop}. \textbf{SyncStart} initializes the blocking set to the current set of readers — the threads that may have already observed any nodes the writer has unlinked. \textbf{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 wait for all overlapping readers to simultaneously be out of critical sections. If two reader threads \(A\) and \(B\) overlap some \textbf{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 \textbf{SyncStart} is used \textit{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 \textbf{ReadBegin} and \textbf{ReadEnd}. \textbf{ReadBegin} simply records the current thread's presence as an active reader. \textbf{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 \textbf{ReadBegin}, \textbf{ReadEnd}, \textbf{SyncStart}, and \textbf{SyncStop}. \textbf{ReadBegin} ensures the set of active readers is known. When a grace period is required, \textbf{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 \textbf{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 modelling 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 be-
  cause 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 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

\[ \tau ::= \text{rcultr } \rho \mathcal{N} \mid \text{rcuFresh } \mathcal{N} \mid \text{unlinked} \mid \text{undef} \mid \text{freeable} \mid \text{rcuRoot} \]
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 $\rho$ from the root. A component $N$ 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 rcuItr 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 may be found in 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 rcuItr type, it has field mappings set $N$. We set the field mappings in the set of an existing rcuFresh reference to be the same as field mappings in the set of rcuItr reference when we replace the heap referenced by rcuItr 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.
\[ N = \{ f_0, \ldots, f_n \mid f_i : \text{ FName} \wedge 0 \leq i \leq n \wedge (y \in \text{ Var} \lor y \in \{ \text{ null} \}) \} \]

\[ N_{f,g} = N \setminus \{ f \rightarrow \_ \} \]

\[ N_{f,g} = \{ \} \]

\[ N(f \rightarrow y) = N \]

\[ N(f \rightarrow x \setminus y) = N \setminus \{ f \rightarrow x \} \cup \{ f \rightarrow y \} \]

\[ \vdash N \ll N' \]

\[ \vdash N_{f,g} \ll N(f \rightarrow y) \]

\[ \vdash N_{f,g} \ll N' \]

\[ \vdash N \ll N' \]

\[ \vdash \rho \ll \rho' \]

\[ \vdash T \ll T' \]

\[ \vdash F \ll F' \]

\[ \vdash \text{ CS(binding)} \]

\[ \vdash \text{ CS(domain)} \]

Figure 3: Sub-Typing Rules

\[ \vdash \text{ RE-index} \]

\[ \vdash \text{ Loop} \]

\[ \vdash \text{ Branch} \]

\[ \vdash \text{ Branch2} \]

\[ \vdash \text{ CS(binding)} \]

\[ \vdash \text{ CS(domain)} \]

Figure 4: Type Rules for Control-Flow.

Figure 3 includes four judgments for subtyping. The first two — \( \vdash N \ll N' \) and \( \vdash T \ll T' \) — describe relaxations of field maps and paths respectively. \( \vdash N \ll N' \) is read as “the field map \( N \) is more precise than \( N' \)” and similarly for paths. The third judgment \( \vdash T \ll T' \) uses path and field map subtyping to give subtyping among \( \text{rcultr} \) types — one \( \text{rcultr} \) is a subtype of another if its paths and field maps are similarly more precise — and to allow \( \text{rcultr} \) references to be “forgotten” — this is occasionally needed to satisfy non-interference checks in the type rules. The final judgment \( \vdash F \ll F' \) 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 ↦ cur in par’s field map, and a similar path disjunction in cur’s path. This is reflected in Figure 5’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 naively 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 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 type system (T-UnlinkH, T-Replace and T-Insert) ensure uses of any indexing variable are never separated — either all are reindexed, or none are.

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; $\Gamma \vdash_R C \rightarrow \Gamma''$ for reader critical sections, and
$\Gamma \vdash_W C \rightarrow \Gamma''$ 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 ($\epsilon$) 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, after the update, cur’s
path is Next ($= \epsilon \cdot $Next), extending the par node’s path by the field read. This
1. Kuru et al.

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 (rcultr), 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 related rules (T-UnlinkH, T-Replace and T-Insert) 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).

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.

$$\forall f \in \text{dom}(N_1). f \neq f_2 \implies (N_1(f) = \text{null})$$

Finally, the rule must ensure that no types in $\Gamma$ are invalidated. This could happen one of two ways: either a field map in $\Gamma$ for an alias of $x$ duplicates the assumption that $x.f_1 = z$ (which is changed by this write), or $\Gamma$ contains a descendant of $r$, whose path from the root will change when its ancestor is modified. The final assumption of $T$-$\text{UNLINKH}$ (the implication) checks that for every $\text{rcultr}$ reference $n$ in $\Gamma$, 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 path is not an extension of $r$ (i.e., it is not a descendant).

$$\forall n \in \Gamma, m, N_3, p_3. f : \text{rcultr} p_3 N_3([f \mapsto m]) \implies \begin{cases} \neg \text{MayAlias}(p_3, \{p, p_1, p_2\}) \\ \forall m \notin \{z, r\} \\ \forall p_4 \neq r. \neg \text{MayAlias}(p_3, p_2, p_4) \end{cases}$$

$\text{MayAlias}$ 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 $\text{MayAlias}$ 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 $\text{rcuFresh}$ reference faces the same aliasing complications as direct unlinking. We illustrate these challenges in Figures 7a and 7b. Figures 31a and 31b in Appendix D illustrating complexities in unlinking). The square $R$ nodes are root nodes, and $H$ nodes are general heap nodes. All resources in red and green form the memory footprint of unlinking. The hollow red circular nodes – $pr$ and $cr$ – point to the nodes involved in replacing $H_1$ (referenced by $cr$) with $H_f$ (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 dashed green $H_f$ denotes the freshly allocated heap node referenced by green dashed $cf$. The dashed green field $l$ is set to point to the referent of $cl$ and the green dashed 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 green — its static type would be $rcu\text{Fresh}$. Note that we cannot duplicate a $rcu\text{Fresh}$ 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 ($T\text{-}\text{Replace}$) prevents path aliasing (nonexistence of $a_0$ and $a_1$) by negating a $May\text{Alias}$ 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 $(H_3, H_2)$ 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 $rcu\text{Fresh}$. This is part of enforcing locality on the heap mutation and captured by assertion $N = N'$ in the type rule ($T\text{-}\text{Replace}$).

**Inserting a Fresh Node** $T\text{-}\text{Insert}$ type checks heap updates that link a fresh node into a linked data structure. Inserting a $rcu\text{Fresh}$ 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 $o$. We prevent this inconsistency with

$$
\forall x \in \Gamma, N_3, \rho_3, f_1, y. (x : rcuItr \rho_3 N_3([f_1 \mapsto y])) \implies (\forall \rho_4 \neq \epsilon. \neg May\text{Alias}(\rho_3, \rho, \rho_4))
$$

There is also another rule, $T\text{-}\text{LinkF-Null}$, 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).

**Entering a Critical Section** *(Referencing inside RCU Blocks)* There are two rules ($ToRCU\text{READ}$ in Figure 6 and $ToRCU\text{WRITE}$ in Figure 6 for moving to RCU typing: one for entering a write region, and one for a read region.*
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]. Appendix C contains the 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, but it motivates some of the conditional-related flexibility discussed in Section 4.2. The use of disjunction (Left|Right) in field maps and paths is required to capture traversals which follow different fields at different times, such as the lookup in a binary search tree.

The most subtle aspect of the deletion is the final step in the case the node $R$ to remove has both children. In this case, the value $V$ of the left-most node of $R$’s right child — the next element in the collection order — is copied into a new node, which is then used to replace node $R$: the replacement’s fields exactly match $R$’s except for the data (T-REPLACE via $N_1 = N_2$), and the parent is updated to reference the replacement, unlinking $R$. At this point, there are two nodes with value $V$ in the tree (weak BST property of the Citrus [3]): the replacement node, and what was the left-most node under $R$’s right child. This latter (original) node for $V$ must be unlinked, which is simplified because by being left-most the left child is null, avoiding another round of replacement (T-UNLINKH via $\forall f \in \text{dom}(N_1) \cdot f \neq f_2 \implies (N_1(f) = \text{null})$. The complexity in checking safety here is that once $R$ is found, another loop is used to find $V$ and its parent (since that node will later be removed as well). After $V$ is found, there are two local unlinking operations, at different depths of the tree. This is why the type system must keep separate abstract iteration counts 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.

6 Soundness

This section outlines the proof of type soundness — our full proof appears in Appendices A.1, A.2, A.3 and A.4. We prove type soundness by embedding the type system into an abstract concurrent separation logic called the Views Framework [11], 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 \cite{17,16}, this consists of several key steps:

1. Define runtime states and semantics for the atomic actions of the language. These are exactly the semantics from Figure 2 in Section 3.

2. Define a denotational interpretation \llbracket - \rrbracket of the types (Figure 5) in terms of an \textit{instrumented} execution state – a runtime state (Section 3) with additional bookkeeping to simplify proofs. The denotation encodes invariants specific to each type, like the fact that unlinked references are unreachable from the heap. The instrumented execution states are also constrained by additional \textit{global WellFormedness invariants} (Appendix A.2) that the type system is intended to maintain, such as tree structure of the data structure.

3. Prove a lemma – called \textit{Axiom Soundness} (Lemma 2) – that the type rules for atomic actions are sound. Specifically, that given a state in the denotation of the pre-type-environment of a primitive type rule, the operational semantics produce a state in the denotation of the post-type-environment. This includes preservation of global invariants.

4. Give a desugaring \llbracket - \rrbracket of non-trivial control constructs (Figure 10) into the simpler non-deterministic versions provided by Views.

The top-level soundness claim then requires proving that every valid source typing derivation corresponds to a valid derivation in the Views logic: \forall \Gamma, C, \Gamma' \vdash_M C \triangleright \Gamma \Rightarrow \llbracket \Gamma \rrbracket \downarrow C \downarrow \llbracket \Gamma' \rrbracket. Because the parameters given to the Views framework ensure the Views logic’s Hoare triples \llbracket - \rrbracket are sound, this proves soundness of the type rules with respect to type denotations. Because 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 Invariants of RCU Views and Denotations of Types

In this section we aim to convey the intuition behind the predicate \textit{WellFormed} which enforces global invariants on logical states, and how it interacts with the denotations of types in key ways. \textit{WellFormed} is the conjunction of a number of more specific invariants, which we outline here. For full details, see Appendix A.2.

\textbf{The Invariant for Read Traversal} Reader threads read 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 \textit{iterator} as they can only use local \texttt{rcultr} references. To this end, a Readers-\textit{Iterators-Only} invariant asserts that a heap location can only be observed as \textit{iterator} by the reader threads.

\textbf{Invariants on Grace-Period} Our logical state (Section 6.2) includes some “free list” auxiliary state tracking which readers are still accessing \textit{each} unlinked node during grace periods. This must be consistent with the bounding thread set...
\( B \) in the machine state. The \textit{Readers-In-Free-List} invariant asserts that all reader threads with observations of unlinked locations are in the to-free lists for those locations. This is essentially tracking which readers are being “shown grace” for each location. The \textit{Iterators-Free-List} invariant complements this by asserting all readers with such observations 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 \texttt{freeable} or \texttt{unlinked}. Once the writer unlinks a heap node, it first observes the heap node as \texttt{unlinked} then \texttt{freeable}. The denotation of \texttt{freeable} is only valid following a grace period: it asserts no readers hold aliases of the \texttt{freeable} reference. The denotation of \texttt{unlinked} permits the either the same (perhaps no readers overlapped) or that it is in the to-free list.

\textbf{Invariants on Safe Traversal against Unlinking} The write-side critical section must guarantee that no updates to the heap cause invalid memory accesses. The \textit{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 \texttt{rcultr} reference, \([\texttt{rcultr} \rho \mathcal{N}]_{\text{tid}}\), asserts that following a path from the root compatible with \( \rho \) reaches the referent, and all are observed as iterator.

Only a bounding thread may view an (unlinked) heap location in the free list as iterator. The denotation of the reader thread’s \texttt{rcultr} reference, \([\texttt{rcultr}]_{\text{tid}}\), requires the referent be either reachable from the root or an unlinked reference in the to-free list. At the same time, it is essential that reader threads arriving after a node is unlinked cannot access it. The invariants \texttt{Unlinked-Reachability} and \texttt{Free-List-Reachability} ensure that any unlinked nodes are reachable only from other unlinked nodes, and never from the root.

\textbf{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 \textit{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 \textit{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. \textit{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 \texttt{rcuFresh} type, \([\texttt{rcuFresh} \mathcal{N}]_{\text{tid}}\), enforces that fields in \( \mathcal{N} \) point to valid heap locations (observed as iterator by the writer thread).

\textbf{Invariants on Tree Structure} Our invariants enforce the tree structure heap layouts for data structures. \textit{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, \textit{Unique-Root} enforces unreachability of the root from any heap location that is reachable from root itself.
6.2 Proof

This section provides more details on how the Views Framework [11] is used to prove soundness, giving the major parameters to the framework and outlining key lemmas. Section 3 defined what Views calls atomic actions (the primitive operations) and their semantics of 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, $\sigma = (s, b, l, rt, R, B)$;
- An observation map, $O$, of type $\text{Loc} \to \mathcal{P}(\text{obs})$
- Undefined variable map, $U$, of type $\mathcal{P}(\text{Var} \times \text{TID})$
- Set of threads, $T$, of type $\mathcal{P}(\text{TIDS})$
- A to-free map (or free list), $F$, of type $\text{Loc} \to \mathcal{P}(\text{TID})$

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.

$$\text{obs} := \text{iterator} \ t\text{id} \ | \ \text{unlinked} \ | \ \text{fresh} \ | \ \text{freeable} \ | \ \text{root}$$

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.

Assertions in the Views logic are (almost) sets of the logical states that satisfy a validity predicate $\text{WellFormed}$, outlined in Section 6.1.

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

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

$$\llbracket - \rrbracket : \text{TypeEnv} \to \text{TID} \to \mathcal{P}(\mathcal{M})$$

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 3.

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:
In order to validate the frame rule in the Views Framework’s logic, the assertions between a local action and framed-away state is the reflexive transitive closure $R \subseteq M \times M$.

WellFormed states are closed under both composition (with another WellFormed state) and interference ($R$ relates WellFormed states only to other WellFormed states).

The framing/weakening type rule will be translated to a use of the frame rule in the Views Framework’s logic. There separating conjunction is simply the existence of two composable instrumented states:

$$m \in P \triangleleft Q \overset{def}{=} \exists m'. \exists m''. m' \in P \land m'' \in Q \land m \in m' \bullet m''$$

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
Alternatively, we say that environment denotation is stable

\( \Gamma \)

ment.

We ellide the statement of the analogous result for the read-side critical section, where

Additionally, interference must distribute over composition:

This can be viewed as 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 \( \forall \Gamma, C, \Gamma' \), \( \Gamma \vdash C \iff \Gamma' \Rightarrow \llbracket \Gamma \rrbracket \mid \llbracket C \rrbracket \mid \llbracket \Gamma' \rrbracket \).

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 10. \( \text{assume}(b) \) is a standard idea in verification semantics [4,30], which “does nothing” (freezes) if the condition \( b \) is false, so its postcondition in the Views logic can reflect the truth of \( b \).

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 [17,16]) address subtyping in a Views-based proof. Views defines the notion of \( \text{view shift} \) 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:

\[
p \sqsubseteq q \overset{\text{def}}{=} \forall m \in M. [p \ast \{m\}] \subseteq [q \ast R(\{m\})]
\]

We are now finally ready to prove the key lemmas of the soundness proof, relating subtyping 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, \( \Gamma_1 \models_M \alpha \vdash \Gamma_2 \), we show \( \forall m. [\alpha](\llbracket \Gamma_1 \rrbracket_{\text{tid}} \ast \{m\}) \subseteq [\llbracket \Gamma_2 \rrbracket_{\text{tid}} \ast R(\{m\})] \)

*Proof.* By case analysis on \( \alpha \). Details in Appendix A.1.

**Lemma 3 (Context-SubTyping-M).** \( \Gamma \prec: \Gamma' \Rightarrow [\llbracket \Gamma \rrbracket_{\text{tid}} \sqsubseteq [\llbracket \Gamma' \rrbracket_{\text{tid}}] \)

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

---

2 This is the same notion present in later program logics like Iris [?], though more recent variants are more powerful.
Lemma 4 (Views Embedding for Write-Side).
\[ \forall \Gamma, C, \Gamma', t. \Gamma \vdash_{M} C \vdash_{M} \Gamma' \Rightarrow [\Gamma]' \cap [M]_t \vdash [C]_t \vdash [\Gamma]' \cap [M]_t \]

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.

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 Related Work

Our type system builds on a great deal of related work on RCU implementations and models; and general concurrent program verification (via program logics, model checking, and type systems).

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, that reader critical sections do not span grace periods. They prove by hand that the Linux kernel RCU implementation [1] satisfies this property. According to the fundamental requirements of RCU [27], our model in Section 3 can be considered as a valid RCU implementation satisfying all requirements for an RCU implementation (assuming sequential consistency) aside from one performance optimization, Read-to-Write Upgrade, which is important in practice but not memory-safety centric:

– Grace-Period and Memory-Barrier Guarantee: To reclaim a heap location, a mutator thread must synchronize with all of the reader threads with overlapping read-side critical sections to guarantee that none of the updates to the memory cause invalid memory accesses. The operational semantics enforce a protocol on the mutator thread’s actions. First it unlinks a node from the data structure; the local type for that reference becomes unlinked. Then it waits for current reader threads to exit, after which the local type is freeable. Finally, it may safely reclaim the memory, after which the local type is undef. The semantics prevent the writer from reclaiming too soon by adding the heap location to the free list of the state, which is checked dynamically by the actual free operation. We discuss the grace period and unlinking invariants in our system in Section 6.1.

– Publish-Subscribe Guarantee: Fresh heap nodes cannot be observed by the reader threads until they are published. As we see in the operational semantics, once a new heap location is allocated it can only be referenced by a local variable of type fresh. Once published, the local type for that reference
becomes $\text{rcultr}$, indicating it is now safe for the reader thread to access it with local references in $\text{rcultr}$ type. We discuss the related type assertions for inserting/replacing (Figures 7a-7b) a fresh node in Section 4.3 and the related invariants in Section 6.1.

- **RCU Primitives Guaranteed to Execute Unconditionally:** Uncoditional execution of RCU Primitives are provided by the definitions in our operational semantics for our RCU primitives (e.g. $\text{ReadBegin}$, $\text{ReadEnd}$, $\text{WriteBegin}$ and $\text{WriteEnd}$) as their executions do not consider failure/retry.

- **Guaranteed Read-to-Write Upgrade:** This is a performance optimization which allows the reader threads to upgrade the read-side critical section to the write-critical section by acquiring the lock after a traversal for a data element and ensures that the upgrading-reader thread exit the read-critical section before calling RCU synchronization. This optimization also allows sharing the traversal code between the critical sections. Read-to-Write is an important optimization in practice but largely orthogonal to memory-safety. Current version of our system provides a strict separation of traverse-and-update and traverse-only intentions through the type system (e.g. different iterator types and rules for the RCU Write/Read critical sections) and the programming primitives. As a future work, we want to extend our system to support this performance optimization.

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 Alglaive et al.’s.

Tassarotti et al. model a well-known way of implementing RCU synchronization without hurting readers’ performance, Quiescent State Based Reclamation (QSRB) [10] where synchronization between the writer thread and reader threads provided via per-thread counters. Tassarotti et al. 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 sequentially-consistency. They require release-writes and acquire-reads to the QSRB counters for proper synchronization in between the mutator and the reader threads. This protocol is exactly what we enforce over the logical observations of the mutator thread: from unlinked to freeable. Tassarotti et al.’s synchronization for linking/publishing new nodes occurs in a similar way to ours, so we anticipate it would be possible to extend our type system in the future for similar weak memory models.

**Program Logics** Fu et al. [15] extend Rely-Guarantee and Separation-Logic [36][13][12] with the past-tense temporal operator to eliminate the need for using a history variable and lift the standard separation conjuction to assert over on execution histories. Gotsman et al. [15] 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 [15] for Michael’s non-blocking stack [29] 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 [?] make a compelling argument that separating memory safety from correctness is profitable, and we provide such a decoupled memory safety argument.

Realizing our RCU Model A direct implementation of our semantics would yield unacceptable performance, since both entering (ReadBegin) and exiting (ReadEnd) modify shared data structures for the bounding-threads and readers sets. A slight variation on our semantics would use a bounding set that tracked such a snapshot of counts, and a vector of per-thread counts in place of the reader set. Blocking grace period completion until the snapshot was strictly older than all current reader counts would be more clearly equivalent to these implementations. Our current semantics are simpler than this alternative, while also equivalent.

Model Checking Kokologiamakis et al. [20] use model-checking to test the core of Tree RCU in Linux kernel. Liang et al. [22] use model-checking to verify the grace period guarantee of Tree RCU. Both focus on validating a particular RCU implementation, whereas we focus on verifying memory safety of clients independent of implementation. Desnoyers et al. [10] use the SPIN model checker to verify a user-mode implementation of RCU and this requires manual translation from C to SPIN modelling language. In addition to being implementation-specific, they require test harness code, validating its behavior rather than real client code.

Type Systems Howard et al. [19,8] present a Haskell library called Monadic RP which provides types and relativistic programming constructs for write/read critical sections which enforce correct usage of relativistic programming pattern. They also have only checked a linked list. They claim handling trees (look-up followed by update) as a future work [19]. Thus our work is the first type system for ensuring correct use of RCU primitives that is known to handle more complex structures than linked lists.

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. [?] 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 [11] 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.

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
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
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
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
5. Calcagno, C., O’Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: Proceedings of the 22Nd Annual IEEE Symposium on Logic in Computer Science. pp. 366–378. LICs ’07, IEEE Computer Society, Washington, DC, USA (2007). https://doi.org/10.1109/LICS.2007.30
6. 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
7. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: Vcc: A practical system for verifying con-
current c. In: Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics. pp. 23–42. TPHOLs ’09, Springer-Verlag, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2

8. 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

9. 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

10. 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

11. 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

12. 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

13. 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

14. 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)

15. 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).

16. 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

17. 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

18. 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
19. 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

20. 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

21. 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

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

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. 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

30. Miller, P., Schwерhoff, 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

31. Paul E. McKenney, Mathieu Desnoyers, L.J., Tripplett, J.: The rcu-barrier menagerie (Nov 2016), https://lwn.net/Articles/573497/

32. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. LICS '02, IEEE Computer Society, Washington, DC, USA (2002), http://dl.acm.org/citation.cfm?id=645683.664578
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](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 â–¢ 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](http://dx.doi.org/10.1007/978-3-540-74407-8_18)
A Complete Soundness Proof of Atoms and Structural Program Statements

A.1 Complete Constructions for Views

To prove soundness we use the Views Framework [11]. The Views Framework takes a set of parameters satisfying some properties, and produces a soundness proof for a static reasoning system for a larger programming language. Among other parameters, the most notable are the choice of machine state, semantics for atomic actions (e.g., field writes, or WriteBegin), and proofs that the reasoning (in our case, type rules) for the atomic actions are sound (in a way chosen by the framework). The other critical pieces are a choice for a partial view of machine states — usually an extended machine state with meta-information — and a relation constraining how other parts of the program can interfere with a view (e.g., modifying a value in the heap, but not changing its type). Our type system will be related to the views by giving a denotation of type environments in terms of views, and then proving that for each atomic action shown in 2 in Section 3 and type rule in Figures 6 Section 4.2 and 32 Appendix E, given a view in the denotation of the initial type environment of the rule, running the semantics for that action yields a local view in the denotation of the output type environment of the rule. The following works through this in more detail. We define logical states, LState to be

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

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.

Each memory region can be observed in one of the following type states within a snapshot taken at any time

\[ \text{obs} := \text{iterator} \mid \text{tid} \mid \text{unlinked} \mid \text{fresh} \mid \text{freeable} \mid \text{root} \]

We are interested in RCU typed of heap domain which we define as:

\[ \text{RCU} = \{ o \mid \text{ftype}(f) = \text{RCU} \land \exists o'. h(o', f) = o \} \]

A thread’s (or scope’s) view of memory is a subset of the instrumented (logical states), which satisfy certain well-formedness criteria relating the physical state and the additional meta-data \( (O, U, T \text{ and } F) \)

\[ \mathcal{M} = \{ m \in (\text{MState} \times O \times U \times T \times F) \mid \text{WellFormed}(m) \} \]
We do our reasoning for soundness over instrumented states and define an erasure relation
\[
[-] : \text{MState} \rightarrow \text{LState}
\]
that projects instrumented states to the common components with MState.

Every type environment represents a set of possible views (well-formed logical states) consistent with the types in the environment. We make this precise with a denotation function
\[
[-] : \text{TypeEnv} \rightarrow \text{TID} \rightarrow \mathcal{P}(\mathcal{M})
\]
that yields the set of states corresponding to a given type environment. This is defined in terms of denotation of individual variable assertions
\[
[- ; -] : \text{Var} \rightarrow \text{Type} \rightarrow \text{TID} \rightarrow \mathcal{P}(\mathcal{M})
\]
The latter is given in Figure 11. To define the former, we first need to state what it means to combine logical machine states.

Composition of instrumented states is an operation
\[
\bullet : \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{M}
\]
that is commutative and associative, and defined component-wise in terms of composing physical states, observation maps, undefined sets, and thread sets as shown in Figure 12. An important property of composition is that it preserves validity of logical states:
Lemma 5 (Well Formed Composition). Any successful composition of two well-formed logical states is well-formed:

$$\forall x, y, z. \text{WellFormed}(x) \implies \text{WellFormed}(y) \implies x \cdot y = z \implies \text{WellFormed}(z)$$

Proof. By assumption, we know that WellFormed(x) and WellFormed(y) hold. We need to show that composition of two well-formed states preserves well-formedness which is to show that for all z such that $x \cdot y = z$, WellFormed(z) holds. Both x and y have components $((s_x, h_x, l_x, r_x, R_x, B_x), O_x, U_x, T_x, F_x)$ and $((s_y, h_y, l_y, r_y, R_y, B_y), O_y, U_y, T_y, F_y)$, respectively. ⊗ operator over stacks $s_x$ and $s_y$ enforces $\text{dom}(s_x) \cap \text{dom}(s_y) = \emptyset$ which enables to make sure that wellformed mappings in $s_x$ does not violate wellformed mappings in $s_y$ when we union these mappings for $s_z$. Same argument applies for $\cdot_F$ operator over $F_x$ and $F_y$. Disjoint unions of wellformed $R_x$ with wellformed $R_y$ and wellformed $B_x$ with wellformed $B_y$ preserves wellformedness in composition as it is disjoint union of different wellformed elements of sets. Wellformed unions of $O_x$ with $O_y$, $U_x$ with $U_y$ and $T_x$ with $T_y$ preserve wellformedness. When we compose $h_x(s(x, tid), f)$ and $h_y(s(x, l), f)$, it is easy to show that we preserve wellformedness if both threads agree on the heap location. Otherwise, if the heap location is undefined for one thread but a value for the other thread then composition considers the value. If a heap location is undefined for both threads then this heap location is also undefined for the location. All the cases for heap composition still preserves the wellformedness from the assumption that x and y are wellformed.

We define separation on elements of type contexts

- For read-side as $[x_1 : T_1, \ldots, x_n : T_n]_{\text{tid}, R} = [x_1 : T_1]_{\text{tid}} \cap \cdots \cap [x_n : T_n]_{\text{tid}} \cap [R]_{\text{tid}}$ where $[R]_{\text{tid}} = \{(s, h, l, r, R, B), O, U, T, F \mid \text{tid} \in R\}$
- For write-side as $[x_1 : T_1, \ldots, x_n : T_n]_{\text{tid}, M} = [x_1 : T_1]_{\text{tid}} \cap \cdots \cap [x_n : T_n]_{\text{tid}} \cap [M]_{\text{tid}}$ where $[M]_{\text{tid}} = \{(s, h, l, r, R, B), O, U, T, F \mid \text{tid} = l\}$
Safe Deferred Memory Reclamation with Types

\[ \llbracket x_1 : T_1, \ldots, x_n : T_n \rrbracket_{\text{tid}, O} = \llbracket x_1 : T_1 \rrbracket_{\text{tid}} \cap \ldots \cap \llbracket x_n : T_n \rrbracket_{\text{tid}} \cap \llbracket O \rrbracket_{\text{tid}} \]

where \( \llbracket O \rrbracket_{\text{tid}} = \{(s, h, l, rt, R, B), O, U, T, F | \text{tid} \neq l \land \text{tid} \notin R\} \).

Partial separating conjunction then simply requires the existence of two states that compose:

\[ m \in P \ast Q \overset{df}{=} \exists m'. \exists m''. m' \in P \land m'' \in Q \land m \in m' \bullet m'' \]

Different threads’ views of the state may overlap (e.g., on shared heap locations, or the reader thread set), but one thread may modify that shared state. The Views Framework restricts its reasoning to subsets of the logical views that are stable with respect to expected interference from other threads or contexts. We define the interference as (the transitive reflexive closure of) a binary relation \( R \) on \( \mathcal{M} \), and a View in the formal framework is then:

\[ \text{View}_{\mathcal{M}} \overset{df}{=} \{ M \in \mathcal{P}(\mathcal{M}) | R(M) \subseteq M \} \]

Thread interference relation

\[ R \subseteq \mathcal{M} \times \mathcal{M} \]

defines permissible interference on an instrumented state. The relation must distribute over composition:

\[ \forall m_1, m_2, m. (m_1 \bullet m_2)Rm \implies \exists m'_1, m'_2. m_1Rm'_1 \land m_2Rm'_2 \land m \in m'_1 \bullet m'_2 \]

where \( R \) is transitive-reflexive closure of \( R_0 \) shown at Figure 12. \( R_0 \) (and therefore \( R \)) also “preserves” validity:

**Lemma 6 (Valid \( R_0 \) Interference).** For any \( m \) and \( m' \), if WellFormed(\( m \)) and \( mR_0m' \), then WellFormed(\( m' \)).

**Proof.** By assumption, we know that \( m = (s, h, l, rt, R, B), O, U, T, F \) is well-formed. We also know that \( m' = (s', h', l', rt', R', B'), O', U', T', F' \) is related to \( m \) via \( R_0 \). By assumptions in \( R_0 \) and semantics, we know that \( O, R, T \) and \( U \) which means that these components do not have any effect on wellformedness of the \( m \). In addition, change on stack, \( s \), does not affect the wellformedness as

\[ \forall x, t \in T. s(x, t) = s'(x, t) \]

Moreover, from semantics we know that \( l \) and \( h \) can only be changed by writer thread and from \( R_0 \)

\[ l \in T \rightarrow (h = h' \land l = l') \]

\[ l \in T \rightarrow F = F' \]

and by assumptions from the lemma(WellFormed(\( m \)).RINFL) we can conclude that \( F, l \) and \( h \) do not have effect on wellformedness of the \( m \).
Lemma 7 (Stable Environment Denotation-M). For any closed environment $\Gamma$ (i.e., $\forall x \in \text{dom}(\Gamma)., \text{FV}(\Gamma(x)) \subseteq \text{dom}(\Gamma)$):

$$R([\Gamma]_{M,\text{tid}}) \subseteq [\Gamma]_{M,\text{tid}}$$

Alternatively, we say that environment denotation is stable (closed under $R$).

Proof. By induction on the structure of $\Gamma$. The empty case holds trivially. In the other case where $\Gamma' = \Gamma', x : T$, we have by the inductive hypothesis that

$$[\Gamma']_{M,\text{tid}}$$

is stable, and must show that

$$[\Gamma']_{M,\text{tid}} \cap [x : T]_{\text{tid}}$$

is as well. This latter case proceeds by case analysis on $T$.

We know that $O, U, T, R, s$ and $rt$ are preserved by $R_0$. By unfolding the type environment in the assumption we know that $\text{tid} = l$. So we can derive conclusion for preservation of $F$ and $h$ and $l$ by

$$l \in T \rightarrow (h = h' \land l = l')$$

$$l \in T \rightarrow F = F'$$

Cases in which denotations, $[x : T]$, touching these $R_0$ preserved maps are trivial to show.

Case 1. - unlinked, undef, rcuFresh$\mathcal{N}$ and freeable trivial.

Case 2. - rcuItr$\rho$.$\mathcal{N}$: All the facts we know so far from $R_0$, $\text{tid} = l$ and additional fact we know from $R_0$:

$$\forall \text{tid}, o. \text{iterator} \text{tid} \in O(o) \rightarrow o \in \text{dom}(h)$$

$$\forall \text{tid}, o. \text{iterator} \text{tid} \in O(o) \rightarrow o \in \text{dom}(h')$$

prove this case.

Case 3. - root: All the facts we know so far from $R_0$, $\text{tid} = l$ and additional fact we know from $R_0$:

$$\forall \text{tid}, o. \text{root} \text{tid} \in O(o) \rightarrow o \in \text{dom}(h)$$

$$\forall \text{tid}, o. \text{root} \text{tid} \in O(o) \rightarrow o \in \text{dom}(h')$$

prove this case.

Lemma 8 (Stable Environment Denotation-R). For any closed environment $\Gamma$ (i.e., $\forall x \in \text{dom}(\Gamma)., \text{FV}(\Gamma(x)) \subseteq \text{dom}(\Gamma)$):

$$R([\Gamma]_{R,\text{tid}}) \subseteq [\Gamma]_{R,\text{tid}}$$

Alternatively, we say that environment denotation is stable (closed under $R$).
Proof. Proof is similar to one for Lemma 7 where there is only one simple case, \([x : \text{rcuItr}]\).

The Views Framework defines a program logic (Hoare logic) with judgments of the form \(\{p\}C\{q\}\) for views \(p\) and \(q\) and commands \(C\). Commands include atomic actions, and soundness of such judgments for atomic actions is a parameter to the framework. The framework itself provides for soundness of rules for sequencing, fork-join parallelism, and other general rules. To prove type soundness for our system, we define a denotation of type judgments in terms of the Views logic, and show that every valid typing derivation translates to a valid derivation in the Views logic:

\[
\forall \Gamma, C, \Gamma', \text{tid}. \Gamma \vdash_{M,R} C \Rightarrow \Gamma' \Rightarrow \{[\Gamma]_{\text{tid}}\}[C]_{\text{tid}}{[\Gamma']_{\text{tid}}}
\]

The antecedent of the implication is a type judgment (shown in Figure 6 Section 4.3, Figure 3 Section 4.1 and Figure 32 Appendix E) and the conclusion is a judgment in the Views logic. The environments are translated to views \((\text{View}_M)\) as previously described. Commands \(C\) also require translation, because the Views logic is defined for a language with 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 13. \(\text{assume}(b)\) is a standard construct in verification semantics [4], which “does nothing” (freezes) if the condition \(b\) is false, so its postcondition in the Views logic can reflect the truth of \(b\). This is also the approach used in previous applications of the Views Framework [17,16].

The framework also describes a useful concept called the view shift operator \(\subseteq\), that describes a way to reinterpret a set of instrumented states as a new set of instrumented states. This operator enables us to define an abstract notion of executing a small step of the program. We express the step from \(p\) to \(q\) with action \(\alpha\) ensuring that the operation interpretation of the action satisfies the specification: \(p \subseteq q \subseteq q \Rightarrow \forall m \in M. [p \ast \{m\}] \subseteq [q \ast R(\{m\})]\). Because the Views framework handles soundness for the structural rules (sequencing, parallel composition, etc.), there are really only three types of proof obligations for us to prove. First, we must prove that the non-trivial command translations (i.e., for conditionals and while loops) embed correctly in the Views logic, which is straightforward. Second, we must show that for our environment subtyping, if
Γ ⊂ Γ′, then \([Γ] ⊆ [Γ′]\). And finally, we must prove that each atomic action’s type rule corresponds to a valid semantic judgment in the Views Framework:

\[ ∀m. [α][([Γ_1]_{tid} * \{m\})] ⊆ ([Γ_2]_{tid} * R(\{m\})) \]

The use of \(∗\) validates the frame rule and makes this obligation akin to an interference-tolerant version of the small footprint property from traditional separation logics [32,5].

**Lemma 9 (Axiom of Soundness for Atoms).** For each axiom, \(Γ_1 ⊢_{RMO} α ⊢ Γ_2\), we must show

\[ ∀m. [α][([Γ_1]_{tid} * \{m\})] ⊆ ([Γ_2]_{tid} * R(\{m\})) \]

**Proof.** By case analysis on the atomic action \(α\) followed by inversion on typing derivation. All the cases proved as different lemmas in Section A.3.

Type soundness proceeds according to the requirements of the Views Framework, primarily embedding each type judgment into the Views logic:

**Lemma 10 (Views Embedding for Read-Side).**

\[ ∀Γ, C, Γ′, t. Γ ⊢_M C ⊢ Γ′ \Rightarrow [Γ]_t ∩ [C]_t ⊢ [Γ′]_t \]

**Proof.** Proof is similar to the one for Lemma 11 except the denotation for type system definition is \([R]_t = \{((s, h, l, rt, R, B), O, U, T, F)|t ∈ R\}\) which shrinks down the set of all logical states to the one that can only be defined by types(rcuItr) in read type system.

**Lemma 11 (Views Embedding for Write-Side).**

\[ ∀Γ, C, Γ′, t. Γ ⊢_M C ⊢ Γ′ \Rightarrow [Γ]_t ∩ [M]_t ⊢ [Γ′]_t ∩ [M]_t \]

**Proof.** Induction on derivation of \(Γ ⊢_M C ⊢ Γ′\) and then inducting on the type of first element of the environment. For the nonempty case, \(Γ″, x : T\) we do case analysis on \(T\). Type environment for write-side actions includes only: rcuItr \(ρ N\), undef, rcuFresh, unlinked, freeable. Denotations of these types include the constraint \(t = l\) and other constraints specific to the type’s denotation. The set of logical state defined by the denotation of the type is subset of intersection of the set of logical states defined by \([M]_t \cap [x : T]_t\) which shrinks down the logical states defined by \([M]_t = \{(s, h, l, rt, R, B), O, U, T, F)|t = l\}\) to the set of logical states defined by denotation \([x : T]_t\).

Because the intersection of the environment denotation with the denotations for the different critical sections remains a valid view, the Views Framework provides most of this proof for free, given corresponding lemmas for the atomic actions \(α\):

\[ ∀α, Γ_1, Γ_2, Γ_1 ⊢_R α ⊢ Γ_2 \Rightarrow \]

\[ ∀m. [α][([Γ_1]_{R,tid} * \{m\})] ⊆ ([Γ_2]_{R,tid} * R(\{m\})) \]
\(\forall \alpha, \Gamma_1, \Gamma_2. \Gamma_1 \vdash_{M} \alpha \rightarrow \Gamma_2 \Rightarrow \forall m. \llbracket \alpha \rrbracket (\llbracket \Gamma_1 \rrbracket M, \text{tid}^* \{m\}) \subseteq \llbracket \Gamma_2 \rrbracket M, \text{tid}^* \mathcal{R}(\{m\})]\)

\(\alpha\) ranges over any atomic command, such as a field access or variable assignment.

Denoting a type environment \(\llbracket \Gamma \rrbracket M, \text{tid}\), unfolding the definition one step, is merely \(\llbracket \Gamma \rrbracket \text{tid} \cap \llbracket M \rrbracket \text{tid}\). In the type system for write-side critical sections, this introduces extra boilerplate reasoning to prove that each action preserves lock ownership. To simplify later cases of the proof, we first prove this convenient lemma.

**Lemma 12 (Write-Side Critical Section Lifting).** For each \(\alpha\) whose semantics does not affect the write lock, if

\(\forall m. \llbracket \alpha \rrbracket (\llbracket \Gamma_1 \rrbracket \text{tid}^* \{m\}) \subseteq \llbracket \Gamma_2 \rrbracket \text{tid}^* \mathcal{R}(\{m\})]\)

then

\(\forall m. \llbracket \alpha \rrbracket (\llbracket \Gamma_1 \rrbracket M, \text{tid}^* \{m\}) \subseteq \llbracket \Gamma_2 \rrbracket M, \text{tid}^* \mathcal{R}(\{m\})]\)

**Proof.** Each of these shared actions \(\alpha\) preserves the lock component of the physical state, the only component constrained by \(\llbracket - \rrbracket M, \text{tid}\) beyond \(\llbracket - \rrbracket \text{tid}\). For the read case, we must prove from the assumed subset relationship that for an arbitrary \(m\):

\(\llbracket \alpha \rrbracket (\llbracket \Gamma_1 \rrbracket \text{tid} \cap \llbracket M \rrbracket \text{tid}^* \{m\}) \subseteq \llbracket \Gamma_2 \rrbracket \text{tid} \cap \llbracket M \rrbracket \text{tid}^* \mathcal{R}(\{m\})]\)

By assumption, transitivity of \(\subseteq\), and the semantics for the possible \(\alpha\)s, the left side of this containment is already a subset of

\(\llbracket \Gamma_2 \rrbracket \text{tid}^* \mathcal{R}(\{m\})]\)

What remains is to show that the intersection with \(\llbracket M \rrbracket \text{tid}\) is preserved by the atomic action. This follows from the fact that none of the possible \(\alpha\)s modifies the global lock.

### A.2 Complete Memory Axioms

1. Ownership invariant in Figure 14 invariant asserts that none of the heap nodes can be observed as undefined by any of those threads.
2. Reader-Writer-Iterators-CoExistence invariant in Figure 15 asserts that if a heap location is not undefined then all reader threads and the writer thread can observe the heap location as iterator or the writer thread can observe heap as fresh, unlinked or freeable.
3. Alias-With-Root invariant in Figure 16 asserts that the unique root location can only be aliased with thread local references through which the unique root location is observed as iterator.
4. Iterators-Free-List invariant in Figure 17 asserts that if a heap location is observed as iterator and it is the free list then the observer thread is in the set of bounding threads.
\[
\text{OW}(\sigma, O, U, T, F) = \begin{cases} \\
\forall o, f, f'. \sigma.h(o, f) = v \land \sigma.h(o', f') = v \\
\land v \in \text{OID} \land \text{FType}(f) = \text{RCU} \implies \\
o = o' \land f = f' \\
\text{unlinked} \in O(o) \\
\text{unlinked} \in O(o') \\
\text{freeable} \in O(o) \\
\text{freeable} \in O(o') \\
\text{fresh} \in O(o) \\
\text{fresh} \in O(o') \\
\end{cases}
\]

Fig. 14: Ownership

\[
\text{RWOW}(\sigma, O, U, T, F) = \begin{cases} \\
\forall x, tid, o. \sigma.s(x, tid) = o \land (x, tid) \notin U \implies \\
\text{iterator} \ tid \in O(o) \\
\neg (\sigma.l = tid \land (\text{unlinked} \in O(o))) \\
\neg (\sigma.l = tid \land \text{freeable} \in O(o)) \\
\neg (\sigma.l = tid \land \text{fresh} \in O(o)) \\
\end{cases}
\]

Fig. 15: Reader-Writer-Iterator-Coexistence-Ownership

\[
\text{AWRT}(\sigma, O, U, T, F) = \{(\forall y, tid. h^*(\sigma.r, \epsilon) = s(y, tid) \implies \text{iterator} \ tid \in O(s(y, tid)))\}
\]

Fig. 16: Alias with Unique Root

\[
\text{IFL}(\sigma, O, U, T, F) = \{\forall tid, o. \text{iterator} \ tid \in O(o) \land \forall T' \subseteq T. \sigma.F([o \mapsto T']) \implies tid \in T'\}
\]

Fig. 17: Iterators-Free-List

\[
\text{ULKR}(\sigma, O, U, T, F) = \begin{cases} \\
\forall o. \text{unlinked} \in O(o) \implies \\
\forall o', f', \sigma.h(o', f') = o \implies \\
\text{unlinked} \in O(o') \lor \\
\text{freeable} \in O(o') \\
\end{cases}
\]

Fig. 18: Unlinked-Reachability

5. Unlinked-Reachability invariant in Figure 18 asserts that if a heap node is observed as unlinked then all heap locations from which you can reach to the unlinked one are also unlinked or in the free list.
6. Free-List-Reachability invariant in Figure [19] asserts that if a heap location is in the free list then all heap locations from which you can reach to the one in the free list are also in the free list.

\[ \text{FLR}(\sigma, O, U, T, F) = \begin{cases} \forall o. F([o \mapsto T]) \implies \\
\{ \forall o', f'. \sigma.h(o', f') = o \implies \\
\{ \exists T' \subseteq T. F([o' \mapsto T']) \} \} \end{cases} \]

Fig. 19: Free-List-Reachability

7. Writer-Unlink invariant in Figure [20] asserts that the writer thread cannot observe a heap location as unlinked.

\[ \text{WULK}(\sigma, O, U, T, F) = \{ \forall o. \text{iterator}\sigma.l \in O(o) \implies \text{unlinked} \notin O(o) \land \text{freeable} \notin O(o) \land \text{undef} \notin O(o) \} \]

Fig. 20: Writer-Unlink

8. Fresh-Reachable invariant in Figure [21] asserts that there exists no heap location that can reach to a freshly allocated heap location together with fact on nonexistence of aliases to it.

\[ \text{FR}(\sigma, O, U, T, F) = \{ \forall \text{tid}, x, o. (\sigma.s(x, \text{tid}) = o \land \text{fresh} \in O(o)) \implies \\
\{ (\forall y, o', \text{tid}'. (h(o', f') \neq o) \lor (s(y, \text{tid}) \neq o) \\
\lor (\text{tid}' \neq \text{tid} \implies s(y, \text{tid}') \neq o) ) \} \} \]

Fig. 21: Fresh-Reachable

9. Fresh-Writer invariant in Figure [22] asserts that heap allocation can be done only by writer thread.

\[ \text{WF}(\sigma, O, U, T, F) = \forall \text{tid}, x, o. (\sigma.s(x, \text{tid}) = o \land \text{fresh} \in O(o)) \implies \text{tid} = \sigma.l \]

Fig. 22: Fresh-Writer
10. Fresh-Not-Reader invariant in Figure 23 asserts that a heap location allocated freshly cannot be observed as unlinked or iterator.

\[ \text{FNR}(\sigma, O, U, T, F) = \forall_o. (\text{fresh} \in O(o)) \implies (\forall_{\text{tid}}. \text{iterator tid} \notin O(o) \land \text{unlinked} \notin O(o)) \]

Fig. 23: Fresh-Not-Reader

11. Fresh-Points-Iterator invariant in Figure 24 states that any field of fresh allocated object can only be set to point heap node which can be observed as iterator (not unlinked or freeable). This invariant captures the fact \( N = N' \) asserted in the type rule for fresh node linking (T-Replace).

\[ \text{FPI}(\sigma, O, U, T, F) = \forall_o. (\text{fresh} \in O(o) \land \exists_{f, o'}. h(o, f) = o') \implies (\forall_{\text{tid}}. \text{iterator tid} \in O(o')) \]

Fig. 24: Fresh-Points-Iterator

12. Writer-Not-Reader invariant in Figure 25 asserts that a writer thread identifier can not be a reader thread identifier.

\[ \text{WNR}(\sigma, O, U, T, F) = \{ \sigma.l \notin \sigma.R \} \]

Fig. 25: Writer-Not-Reader

13. Readers-Iterator-Only invariant in the Figure 26 asserts that a reader threads can only make iterator observation on a heap location.

\[ \text{RITR}(\sigma, O, U, T, F) = \{ \forall_{\text{tid} \in \sigma.R.o}. \text{iterator tid} \in O(o) \} \]

Fig. 26: Readers-Iterator-Only

14. Readers-In-Free-List invariant in Figure 27 asserts that for any mapping from a location to a set of threads in the free list we know the fact that this set of threads is a subset of bounding threads (which itself is subset of reader threads).
\[ \text{RINFL}(\sigma, O, U, T, F) = \{ \forall_{o.} F([o \mapsto T]) \implies T \subseteq \sigma.B \} \]

Fig. 27: Readers-In-Free-List

15. Heap-Domain invariant in the Figure 28 defines the domain of the heap.

\[ \text{HD}(\sigma, O, U, T, F) = \forall_{o, f', o'}. \sigma.h(o, f) = o' \implies o' \in \text{dom}(\sigma.h) \]

Fig. 28: Heap-Domain

16. Unique-Root invariant in Figure 29 asserts that a heap location which is observed as root has no incoming edges from any nodes in the domain of the heap and all nodes accessible from root is is observed as iterator. This invariant is part of enforcement for acyclicity.

\[ \text{UNQRT}(\sigma, O, U, T, F) = \{ \forall_{\rho \neq \epsilon}. \text{iterator tid} \in O(h^*(\sigma.rt, \rho)) \wedge \neg (\exists f'. \sigma.rt = h(h^*(\sigma.rt, \rho), f')) \} \]

Fig. 29: Unique-Root

17. Unique-Reachable invariant in Figure 30 asserts that every node is reachable from root node with an unique path. This invariant is a part of acyclicity(tree structure) enforcement on the heap layout of the data structure.

\[ \text{UNQR}(\sigma, O, U, T, F) = \{ \forall_{\rho, \rho'} h^*(\sigma.rt, \rho) \neq h^*(\sigma.rt, \rho') \implies \rho \neq \rho' \} \]

Fig. 30: Unique-Reachable

Each of these memory invariants captures different aspects of validity of the memory under RCU setting, \text{WellFormed}(\sigma, O, U, T, F), is defined as conjunction of all memory axioms.

A.3 Soundness Proof of Atoms

In this section, we do proofs to show the soundness of each type rule for each atomic actions.
Lemma 13 (Unlink).

\[
\llbracket x.f_1 := r \rrbracket ([\llbracket \Gamma, x : \text{rcultr } \rho N([f_1 \rightarrow z]), z : \text{rcultr } \rho' N'([f_2 \rightarrow r]), \ r : \text{rcultr } \rho'' N''][M, \text{tid} \star \{m\}] ) \subseteq \llbracket \Gamma, x : \text{rcultr } \rho N(f_1 \rightarrow z \setminus r), \ z : \text{unlinked}, \ r : \text{rcultr } \rho' N' \rbracket \ast \mathcal{R}([m])
\]

Proof. We assume

(\sigma, O, U, T, F) \in \llbracket \Gamma, x : \text{rcultr } \rho N, \ z : \text{rcultr } \rho' N', \ r : \text{rcultr } \rho'' N'' \rrbracket_{M, \text{tid} \star \{m\}} \tag{1}

WellFormed(\sigma, O, U, T, F) \tag{2}

From assumptions in the type rule of T-UnlinkH we assume that

\[\rho.f_1 = \rho' \text{ and } \rho'.f_2 = \rho'' \text{ and } N(f_1) = z \text{ and } N'(f_2) = r\tag{3}\]

\[\forall f \in \text{dom}(N''). f \neq f_2 \implies N'(f) = \text{null}\tag{4}\]

\[\forall n \in \Gamma, m, N'''', \rho'''', f, n : \text{rcultr } \rho''' N'''([f \rightarrow m]) \implies ((\neg \text{MayAlias}(\rho'', \{\rho, \rho', \rho''\})) \land (m \not\in \{z, r\})) \land (\forall \rho'''', \rho\neq \rho'' : \neg \text{MayAlias}(\rho'', \rho', \rho'''))\tag{5}\]

We split the composition in \[\llbracket \text{as}

(\sigma_1, O_1, U_1, T_1, F_1) \in \llbracket \Gamma, x : \text{rcultr } \rho N, \ z : \text{rcultr } \rho' N', \ r : \text{rcultr } \rho'' N'' \rrbracket_{M, \text{tid}} \tag{6}

(\sigma_2, O_2, U_2, T_2, F_2) = m \tag{7}

\sigma_1 \bullet \sigma_2 = \sigma \tag{8}

O_1 \bullet_O O_2 = O \tag{9}

U_1 \cup U_2 = U \tag{10}

T_1 \cup T_2 = T \tag{11}

F_1 \uplus F_2 = F \tag{12}

WellFormed(\sigma_1, O_1, U_1, T_1, F_1) \tag{13}

WellFormed(\sigma_2, O_2, U_2, T_2, F_2) \tag{14}
We must show \( \exists \sigma'_1, \sigma'_2, O'_1, O'_2, U'_1, U'_2, T'_1, T'_2, F'_1, F'_2 \) such that
\[
(\sigma'_1, O'_1, U'_1, T'_1, F'_1) \in \llbracket \Gamma, x : rcu \rho. N([f \mapsto r]), z : \text{unlinked}, r : rcu \rho'. N'' \rrbracket_{M, tid}
\]
(15)
\[
N(f_1) = r
\]
(16)
\[
(\sigma'_2, O'_2, U'_2, T'_2, F'_2) \in \mathcal{R}([m])
\]
(17)
\[
\sigma'_1 \circ \sigma'_2 = \sigma'
\]
(18)
\[
O'_1 \circ O'_2 = O'
\]
(19)
\[
U'_1 \cup U'_2 = U'
\]
(20)
\[
T'_1 \cup T'_2 = T'
\]
(21)
\[
F'_1 \uplus F'_2 = F'
\]
(22)
\[
\text{WellFormed}(\sigma'_1, O'_1, U'_1, T'_1, F'_1)
\]
(23)
\[
\text{WellFormed}(\sigma'_2, O'_2, U'_2, T'_2, F'_2)
\]
(24)
\[
\text{WellFormed}(\sigma'_1, O'_1, U'_1, T'_1, F'_1)
\]
(25)

We also know from operational semantics that the machine state has changed as
\[
\sigma'_1 = \sigma_1[h(s(x, tid), f_1) \mapsto s(r, tid)]
\]
(26)
and (26) is determined by operational semantics.

The only change in the observation map is on \( s(y, tid) \) from iterator \( tid \) to unlinked
\[
O'_1 = O_1(s(y, tid))[\text{iterator } tid \mapsto \text{unlinked}]
\]
(27)
\[\text{28}\]
follows from (4)

\[
T_1 = \{tid\} \text{ and } tid = \sigma.l
\]
(28)
\[
\sigma'_1 \text{ is determined by operational semantics. The undefined map, free list and } T_1 \text{ need not change so we can pick } U'_1 \text{ as } U_1, T'_1 \text{ as } T_1 \text{ and } F'_1 \text{ as } F_1.
\]
Assuming [6] and choices on maps makes \((\sigma'_1, O'_1, U'_1, T'_1, F'_1)\) in denotation
\[
\llbracket \Gamma, x : rcu \rho. N([f \mapsto r]), z : \text{unlinked}, r : rcu \rho'. N'' \rrbracket_{M, tid}
\]

In the rest of the proof, we prove (24) (25) and show the composition of \((\sigma'_1, O'_1, U'_1, T'_1, F'_1)\) and \((\sigma'_2, O'_2, U'_2, T'_2, F'_2)\). To prove (24) we need to show that each of the memory axioms in Section A.2 holds for the state \((\sigma', O'_1, U'_1, T'_1, F'_1)\).

Let \( o_x \) be \( \sigma.s(x, tid) \), \( o_y \) be \( \sigma.s(y, tid) \) and \( o_z \) be \( \sigma.s(z, tid) \).

**Case 4. - UNQR** [29] and [30] follow from framing assumption [35], denotations of the precondition [6] and [13] UNQR

\[
\rho \neq \rho' \neq \rho''
\]
(29)
and
\[
o_x \neq o_y \neq o_z
\]
(30)
where \( a_x, a_y \) and \( a_z \) are equal to \( \sigma.h^*(\sigma.rt, \rho), \sigma.h^*(\sigma.rt, \rho, f_1) \) and \( \sigma.h^*(\sigma.rt, \rho, f_1, f_2) \) respectively and they \((a_x, a_y, a_z, \rho, \rho')\) are unique.

We must prove

\[
h''(\sigma.rt, \rho) \neq h''(\sigma.rt, \rho, f_1) \implies \rho \neq \rho.f_1
\]  

(31)

to show that uniqueness is preserved.

We know from operational semantics that root has not changed so

\( \sigma.rt = \sigma'.rt \)

From denotations (15) we know that all heap locations reached by following \( \rho \) and \( \rho.f_1 \) are observed as \( \text{iterator tid} \) including the final reached heap locations \( \text{iterator tid} \in O'_1(\sigma'.h^*(\sigma.rt, \rho)) \) and \( \text{iterator tid} \in O'_1(\sigma'.h^*(\sigma.rt, \rho.f_1)) \).

\( \text{unlinked} \in O'_1(o_y) \) follows from 27 and 26 which makes path \( \rho.f_1.f_2 \) invalid(from denotation(12), all heap locations reaching to \( O'_1(o_r) \) from root(\( \sigma.rt \)) are observed as \( \text{iterator tid} \)) so this proves that \( \text{unlinked} \in O'_1(o_y) \) cannot be observed on the path to the \( o_r \) which implies that \( f_2 \) cannot be part of the path and uniqueness of the paths to \( o_x \) and \( o_r \) is preserved. So we conclude 32 and 33

\[
\rho \neq \rho'.
\]  

(32)

\[
a_x \neq a_y \neq a_z
\]  

(33)

from which (31) follows.

Case 5. - **OW** By 13, OW, 26, 27

Case 6. - **RWOW** By 13, RWOW, 26 and 27

Case 7. - **IFL** By 13, WULK, 13, RINFL, 13, IFL, 27 and choice of \( F'_1 \).

Case 8. - **FLR** By choice of \( F'_1 \) and 13

Case 9. - **WULK** By 13, 27 and 28

Case 10. - **WF, FPI and FR** Trivial.

Case 11. - **AWRT** By 13

Case 12. - **HD** By 24, OW(proved), 13, HD and 26

Case 13. - **WNR** By 13, WNR, 24, 27 and 28

Case 14. - **RINFL** By 15, 13, RINFL, choice of \( F'_1 \) and 26
Case 15. - ULKR We must prove 33

\[ \forall o', h', o. h(o', f') = o_y \implies \text{unlinked} \in O'_1(o') \]

\[ \lor (\text{freeable} \in O'_1(o')) \]

(34)

which follows from 13, 13, OW, operational semantics 26 and 27. If o' were observed as iterator then that would conflict with UNQR.

Case 16. - UNQRT: By 13, UNQRT, 27 and 26

To prove 13 we need to show interference relation

\[(\sigma, O_2, U_2, T_2, F_2) \mathcal{R}(\sigma', O'_2, U'_2, T'_2, F'_2)\]

which by definition means that we must show

\[\sigma_2.l \in T_2 \rightarrow (\sigma_2, h = \sigma_2.h \land \sigma_2.l = \sigma_2.l)\]

\[l \in T_2 \rightarrow F_2 = F'_2\]

\[\forall \text{tid}, o. \text{iterator} \text{tid} \in O_2(o) \rightarrow o \in \text{dom}(\sigma_2, h)\]

\[\forall \text{tid}, o. \text{iterator} \text{tid} \in O_2(o) \rightarrow o \in \text{dom}(\sigma_2, h)\]

\[O_2 = O'_2 \lor U_2 = U'_2 \land T_2 = T'_2 \land \sigma_2.R = \sigma_2', R \land \sigma_2, rt = \sigma_2', rt\]

\[\forall x, t \in T_2. \sigma_2.s(x, t) = \sigma_2'.s(x, t)\]

\[\forall \text{tid}, o. \text{root} \text{tid} \in O(o) \rightarrow o \in \text{dom}(h)\]

\[\forall \text{tid}, o. \text{root} \text{tid} \in O(o) \rightarrow o \in \text{dom}(h')\]

(40)

(41)

(42)

To prove all relations 35, 40 we assume 28 which is to assume \(T_2\) as subset of reader threads. Let \(\sigma'_2\) be \(\sigma_2\). \(O_2\) need not change so we pick \(O'_2\) as \(O_2\). Since \(T_2\) is subset of reader threads, we pick \(T_2\) as \(T'_2\). We pick \(F'_2\) as \(F_2\).

35 and 40 follow from 28 and choice of \(F'_2\). 11, 52 and 59 are determined by choice of \(\sigma'_2\), operational semantic and choices made on maps related to the assertions.

By assuming 11 we show 25, 37 and 38 follow trivially. 40 follows from choice of \(\sigma'_2\). 26 and 28

To prove 40 consider two cases: \(O'_1 \cap O'_2 = \emptyset\) and \(O'_1 \cap O'_2 \neq \emptyset\). The first case is trivial. The second case is where we consider

\[\text{iterator} \text{tid} \in O'_2(o_y)\]

We also know from 27 that

\[\text{unliked} \in O'_1(o_y)\]

Both together with 9 and 13 proves 20.

To show 13 we consider two cases: \(\sigma'_1.h \cap \sigma'_2.h = \emptyset\) and \(\sigma'_1.h \cap \sigma'_2.h \neq \emptyset\). First is trivial. Second follows from OW-HD and OW-HD. 24, 24 and 23 are trivial by choices on related maps and semantics of composition operations on them. All compositions shown let us to derive conclusion for \((\sigma'_1, O'_1, U'_1, T'_1, F'_1) \bullet (\sigma'_2, O'_2, U'_2, T'_2, F'_2)\).
Lemma 14 (Replace).
\[ \llbracket p.f := n \rrbracket \llbracket \Gamma, p : \text{rcultr}\, \rho\, N, r : \text{rcultr}\, \rho'\, N', n : \text{rcuFresh}\, N'' \rrbracket_{M,tid} \ast \{m\} \rrbracket \subseteq \llbracket \Gamma, p : \text{rcultr}\, \rho\, N([f \mapsto r \setminus n]), n : \text{rcultr}\, \rho'\, N', r : \text{unlinked} \ast \mathcal{R}\{\{m\}\} \rrbracket \]

Proof. We assume
\( (\sigma, O, U, T, F) \in \llbracket \Gamma, p : \text{rcultr}\, \rho\, N, r : \text{rcultr}\, \rho'\, N', n : \text{rcuFresh}\, N'' \rrbracket_{M,tid} \ast \{m\} \)

WellFormed(\( \sigma, O, U, T, F \))

From assumptions in the type rule of T-Replace we assume that
\[ \text{FV}(\Gamma) \cap \{p, r, n\} = \emptyset \] (45)
\[ \rho.f = \rho' \text{ and } N(f) = r \] (46)
\[ N' = N'' \] (47)
\[ \forall x \in \Gamma, N''', \rho'', f', y. \ (x : \text{rcultr}\, \rho''\, N'''([f' \mapsto y])) \implies (\neg \text{MayAlias}(\rho'', \{\rho, \rho'\}) \land (y \neq o)) \] (48)

We split the composition in (43) as
\( (\sigma_1, O_1, U_1, T_1, F_1) \in [\Gamma, p : \text{rcultr}\, \rho\, N, r : \text{rcultr}\, \rho'\, N', n : \text{rcuFresh}\, N'']_{M,tid} \)

(50)
\( (\sigma_2, O_2, U_2, T_2, F_2) = m \)

(51)
\( O_1 \cdot O = O \)

(52)
\( \sigma_1 \cdot \sigma_2 = \sigma \)

(53)
\( U_1 \cup U_2 = U \)

(54)
\( T_1 \cup T_2 = T \)

(55)
\( F_1 \uplus F_2 = F \)

(56)
WellFormed(\( \sigma_1, O_1, U_1, T_1, F_1 \))

(57)
WellFormed(\( \sigma_2, O_2, U_2, T_2, F_2 \))
We must show \( \exists \sigma'_1, \sigma'_2, O'_1, O'_2, U'_1, U'_2, T'_1, T'_2, F'_1, F'_2 \) such that

\[
(\sigma'_1, O'_1, U'_1, T'_1, F'_1) \in [p : \text{rcultr } \rho \mathcal{N}, n : \text{rcultr } \rho' \mathcal{N}'', r : \text{unlinked}, \Gamma]_{M,tid}\]

(58)

\( \mathcal{N}(f) = n \)

(59)

\( (\sigma'_2, O'_2, U'_2, T'_2, F'_2) \in \mathcal{R}(\{m\}) \)

(60)

\( O'_1 \otimes O'_2 = O' \)

(61)

\( \sigma'_1 \otimes \sigma'_2 = \sigma'' \)

(62)

\( U'_1 \cup U'_2 = U' \)

(63)

\( T'_1 \cup T'_2 = T' \)

(64)

\( F'_1 \uplus F'_2 = F' \)

(65)

WellFormed(\( \sigma'_1, O'_1, U'_1, T'_1, F'_1 \))

(66)

WellFormed(\( \sigma'_2, O'_2, U'_2, T'_2, F'_2 \))

(67)

We also know from operational semantics that the machine state has changed as

\[
\sigma'_1 = \sigma_1[h(s(p,tid), f) \mapsto s(n, tid)]
\]

(68)

\( 59 \) is determined directly from operational semantics.

We know that changes in observation map are

\[
O'_1 = O_1(s(r,tid))[\text{iterator } tid \mapsto \text{unlinked}]
\]

(69)

and

\[
O'_1 = O_1(s(n,tid))[\text{fresh } \mapsto \text{iterator } tid]
\]

(70)

\( 71 \) follows from \( 43 \)

\( T_1 = \{tid\} \) and \( tid = \sigma.l \)

(71)

Let \( T'_1 \) be \( T_1 \), \( F'_1 \) be \( F_1 \) and \( \sigma'_1 \) be determined by operational semantics. The undefined map need not change so we can pick \( U'_1 \) as \( U_1 \). Assuming \( 49 \) and choices on maps makes \( (\sigma'_1, O'_1, U'_1, T'_1) \) in denotation

\[
[p : \text{rcultr } \rho \mathcal{N}(f \rightarrow r \setminus n), n : \text{rcultr } \rho' \mathcal{N}'', r : \text{unlinked}, \Gamma]_{M,tid}
\]

In the rest of the proof, we prove \( 66 \) \( 67 \) and show the composition of \( (\sigma'_1, O'_1, U'_1, T'_1, F'_1) \) and \( (\sigma'_2, O'_2, U'_2, T'_2, F'_2) \). To prove \( 68 \) we need to show that each of the memory axioms in Section \( \ref{A.2} \) holds for the state \( (\sigma', O'_1, U'_1, T'_1, F'_1) \).

**Case 17. - UNQR** Let \( o_p \) be \( \sigma.s(p,tid) \), \( o_r \) be \( \sigma.s(r,tid) \) and \( o_n \) be \( \sigma.s(n,tid) \). \( 71 \) and \( 73 \) follow from framing assumption \( 45,48 \), denotations of the precondition \( 19, \ref{pr} \) and \( 28 \) UNQR

\[
\rho \neq \rho.f \neq \forall \mathcal{N}'(f, \rightarrow x_i), \rho.f.f_i
\]

(72)
and

\[ o_p \neq o_r \neq o_n \neq o_i \quad \text{where} \quad o_i = h(o_r, f_i) \quad (73) \]

where \( o_p, o_r \) are \( \sigma.h^*(\sigma.rt, \rho) \), \( \sigma.h^*(\sigma.rt, \rho.f) \) respectively and they(heap locations in 73 and paths in 72) are unique.(From \text{FR}, we assume that there exists no field alias/path alias to heap location freshly allocated \( o_n \)).

We must prove

\[ \rho \neq \rho.f \neq \rho.f.f_i \iff \sigma'.h^*(\sigma.rt, \rho) \neq \sigma'.h^*(\sigma.rt, \rho.f) \neq \sigma'.h^*(\sigma.rt, \rho.f.f_i) \quad (74) \]

We know from operational semantics that root has not changed so

\( \sigma.rt = \sigma'.rt \)

From denotations (58) we know that all heap locations reached by following \( \rho \) and \( \rho.f \) are observed as \text{iteartor tid} including the final reached heap locations(\text{iteratortid} \in O'(\sigma'.h^*(\sigma.rt, \rho)), \text{iteratortid} \in O'(\sigma'.h^*(\sigma.rt, \rho.f)) and \text{iteratortid} \in O'(\sigma'.h^*(\sigma.rt, \rho.f.f_i))).

The preservation of uniqueness follows from (56, 70, 68 and 56) \text{FR}.

from which we conclude 75 and 76

\[ \rho \neq \rho.f \neq \rho.f.f_i \quad (75) \]

\[ o_p \neq o_n \neq o_r \quad (76) \]

from which 74 follows.

\textbf{Case 18. - OW} By 56, 68, 69 and 70.

\textbf{Case 19. - RWOW} By 56, 68, 69 and 70.

\textbf{Case 20. - AWRT} Trivial.

\textbf{Case 21. - IFL} By 56, 68, 69 choice of \( F'_1 \) and operational semantics.

\textbf{Case 22. - FLR} By choice of \( F'_1 \) and 56.

\textbf{Case 23. - FPI} By 58.

\textbf{Case 24. - WULK} Determined by operational semantics By 56, 68, 69, 70 and operational semantics.

\textbf{Case 25. - WF} and \textbf{FR} Trivial.

\textbf{Case 26. - HD}

\textbf{Case 27. - WNR} By 71 and operational semantics.

\textbf{Case 28. - RINFL} Determined by operational semantics 58 and 56.
Case 29. - ULKR We must prove

$$\forall o, f', o', h(o', f') = o_r \implies \text{unlinked} \in O'_1(o')$$

freeable \in O'_1(o') \quad (77)

which follows from 55, 56, OW and determined by operational semantics(68), 69, 72. If o' were observed as iterator then that would conflict with 66. UNQR.

Case 30. - UNQR T By 56. UNQR T, 69, 70 and 68.

To prove 60, we need to show interference relation

$$(\sigma, O_2, U_2, T_2, F_2)R(\sigma', O'_2, U'_2, T'_2, F'_2)$$

which by definition means that we must show

$$\sigma_2. l \in T_2 \implies (\sigma_2. h = \sigma'_2. h \land \sigma_2. l = \sigma'_2. l)$$

$$l \in T_2 \implies F_2 = F'_2$$

$$\forall \text{tid, o. iterator tid} \in O_2(o) \implies o \in \text{dom}(\sigma_2. h)$$

$$\forall \text{tid, o. iterator tid} \in O_2(o) \implies o \in \text{dom}(\sigma'_2. h)$$

$$O_2 = O'_2 \land U_2 = U'_2 \land T_2 = T'_2 \land \sigma_2.R = \sigma'_2.R \land \sigma_2.rt = \sigma'_2.rt$$

$$\forall x, t \in T_2, \sigma_2.s(x, t) = \sigma'_2.s(x, t)$$

$$\forall \text{tid, o. root tid} \in O(o) \implies o \in \text{dom}(h)$$

$$\forall \text{tid, o. root tid} \in O(o) \implies o \in \text{dom}(h')$$

To prove all relations (78-83) we assume 71 which is to assume T_2 as subset of reader threads. Let \sigma'_2 be \sigma_2, F'_2 be F_2. O_2 need not change so we pick O'_2 as O_2. Since T_2 is subset of reader threads, we pick T_2 as T'_2. By assuming 57 we show 67, 80 and 81 follow trivially. 82, 84 and 85 are determined by choice of \sigma'_2, operational semantics and choices made on maps related to the assertions.

To prove 71 consider two cases: O'_1 \cap O'_2 = \emptyset and O'_1 \cap O'_2 \neq \emptyset. The first case is trivial. The second case is where we consider 86 and 87.

$$\text{iterator tid} \in O'_2(o_r) \quad (86)$$

From 69 we know that

$$\text{unliked} \in O'_1(o_r)$$

Both together with 51 and 58 proves 61.

For case 87

$$\text{fresh} \in O_2(o_n) \quad (87)$$

From 70 we know that

$$\text{iterator tid} \in O'_1(o_n)$$
Both together with 51 and 58 proves 61.

To show 62 we consider two cases: \( \sigma'_1 \cap \sigma'_2 = \emptyset \) and \( \sigma'_1 \cap \sigma'_2 \neq \emptyset \). First is trivial. Second follows from 66.

OW-OW-HD and 67.

63, 65 and 64 are trivial by choices on related maps and semantics of the composition operators for these maps. All compositions shown let us to derive conclusion for \((\sigma'_1, O'_1, U'_1, T'_1, F'_1) \circ (\sigma'_2, O'_2, U'_2, T'_2, F'_2)\).

**Lemma 15 (Insert).**

\[
\llbracket p.f := n \rrbracket (\llbracket \Gamma, p : \text{rcultr} \rho \mathcal{N}, r : \text{rcultr} \rho_1 \mathcal{N}_2, n : \text{rcuFresh} \mathcal{N}_1 \rrbracket_{M, \text{tid}} * \{m\}) \subseteq \llbracket \Gamma, p : \text{rcultr} \rho \mathcal{N}( [f \mapsto r \setminus n] ), n : \text{rcultr} \rho_1 \mathcal{N}_1, r : \text{rcultr} \rho_2 \mathcal{N}_2 \rrbracket * \mathcal{R}( \{m\} ) \]

**Proof.** We assume

\[
(\sigma, O, U, T, F) \in \llbracket \Gamma, p : \text{rcultr} \rho \mathcal{N}, r : \text{rcultr} \rho_1 \mathcal{N}_2, n : \text{rcuFresh} \mathcal{N}_1 \rrbracket_{M, \text{tid}} * \{m\}
\]

WellFormed(\( \sigma, O, U, T, F \))

From assumptions in the type rule of T-INSERT we assume that

\[
\text{FV}(\Gamma) \cap \{p, r, n\} = \emptyset \quad (90)
\]
\[
\rho.f = \rho_1 \quad \text{and} \quad \rho.f_4 = \rho_2 \quad \text{and} \quad \mathcal{N}(f) = r \quad (91)
\]
\[
\mathcal{N}(f) = \mathcal{N}_1(f_4) \quad \text{and} \quad \forall f_2 \in \text{dom}(\mathcal{N}_1). f_4 \neq f_2 \implies \mathcal{N}_1(f_2) = \text{null} \quad (92)
\]
\[
\forall x \in \Gamma. \mathcal{N}_3, \rho_3, f_1, y. (x : \text{rcultr} \rho_3 \mathcal{N}_1([f_1 \mapsto y])) \implies (\forall \rho_4 \neq . \neg \text{MayAlias}(\rho_3, \rho.\rho_4)) \quad (93)
\]

We split the composition in 88 as

\[
(\sigma_1, O_1, U_1, T_1, F_1) \in \llbracket \Gamma, p : \text{rcultr} \rho \mathcal{N}, r : \text{rcultr} \rho_1 \mathcal{N}_2, n : \text{rcuFresh} \mathcal{N}_1 \rrbracket_{M, \text{tid}}
\]

(94)
\[
(\sigma_2, O_2, U_2, T_2, F_2) = m
\]

(95)
\[
O_1 \circ O_2 = O
\]

(96)
\[
\sigma_1 \circ \sigma_2 = \sigma
\]

(97)
\[
U_1 \cup U_2 = U
\]

(98)
\[
T_1 \cup T_2 = T
\]

(99)
\[
F_1 \uplus F_2 = F
\]

(100)
\[
\text{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1)
\]

(101)
\[
\text{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2)
\]

(102)
We must show \( \exists \sigma'_1, \sigma'_2, O'_1, O'_2, U'_1, U'_2, T'_1, T'_2, F'_1, F'_2 \) such that

\[
(\sigma'_1, O'_1, U'_1, T'_1, F'_1) \in [p : \text{rcultr } \rho \mathcal{N}(\left[ f \mapsto r \ \backslash \ n \right]), n : \text{rcultr } \rho_1 N_1, r : \text{rcultr } \rho_2 N_2, \Gamma]_{M,tid} \tag{103}
\]

\[
(\sigma'_2, O'_2, U'_2, T'_2, F'_2) \in \mathcal{R}(\{m\}) \tag{104}
\]

\[
O'_1 \bullet_{O} O'_2 = O' \tag{105}
\]

\[
\sigma'_1 \bullet, \sigma'_2 = \sigma' \tag{106}
\]

\[
U'_1 \cup U'_2 = U' \tag{107}
\]

\[
T'_1 \cup T'_2 = T' \tag{108}
\]

\[
F'_1 \cup F'_2 = F' \tag{109}
\]

\[
\text{WellFormed}(\sigma'_1, O'_1, U'_1, T'_1, F'_1) \tag{110}
\]

\[
\text{WellFormed}(\sigma'_2, O'_2, U'_2, T'_2, F'_2) \tag{111}
\]

We also know from operational semantics that the machine state has changed as

\[
\sigma'_1 = \sigma_1[h(s(p, \text{tid}), f) \mapsto s(n, \text{tid})]\tag{112}
\]

We know that changes in observation map are

\[
O'_1 = O_1(s(n, \text{tid}))[\text{fresh} \mapsto \text{iterator } \text{tid}] \tag{113}
\]

\[
\text{114 follows from 88}
\]

\[
T_1 = \{\text{tid}\} \text{ and } \text{tid} = \sigma.l \tag{114}
\]

Let \( T'_1 \) be \( T_1 \), \( F'_1 \) be \( F_1 \) and \( \sigma'_1 \) be determined by operational semantics. The undefined map need not change so we can pick \( U'_1 \) as \( U_1 \). Assuming \( 33 \) and choices on maps makes \((\sigma'_1, O'_1, U'_1, T'_1)\) in denotation

\[
[p : \text{rcultr } \rho \mathcal{N}(f \mapsto r \ \backslash \ n), n : \text{rcultr } \rho_1 N_1, r : \text{rcultr } \rho_2 N_2, \Gamma]_{M,tid}
\]

In the rest of the proof, we prove 110 111 and show the composition of \((\sigma'_1, O'_1, U'_1, T'_1, F'_1)\) and \((\sigma'_2, O'_2, U'_2, T'_2, F'_2)\). To prove 111 we need to show that each of the memory axioms in Section A.2 holds for the state \((\sigma', O'_1, U'_1, T'_1, F'_1)\).

Proofs for \( \text{OW} \), \( \text{RLOW} \), \( \text{AWRT} \), \( \text{IFL} \), \( \text{WULK} \), \( \text{FLR} \), \( \text{FPI} \), \( \text{WF} \), \( \text{FR} \), \( \text{HD} \), \( \text{WNR} \), \( \text{RINFL} \) and \( \text{ULKR} \). The proof of \( \text{UNQR} \) is similar to the ones we did for Lemma 14 and Lemma 15 with a simpler fact to prove: we assume framing conditions 90-93 together with the 101 \( \text{UNQR} \) and 101 \( \text{FR} \) which makes \( \text{110 UNQR} \) trivial.

To prove 104 we need to show interference relation

\[
(\sigma, O_2, U_2, T_2, F_2) \mathcal{R}(\sigma'_1, O'_2, U'_2, T'_2, F'_2)
\]
which by definition means that we must show

\[
\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma'_2.h \land \sigma_2.l = \sigma'_2.l)
\] (115)

\[
l \in T_2 \rightarrow F_2 = F'_2
\] (116)

\[
\forall tid, o. \text{ iterator } tid \in O_2(o) \rightarrow o \in \text{dom}(\sigma_2.h)
\] (117)

\[
\forall tid, o. \text{ iterator } tid \in O_2(o) \rightarrow o \in \text{dom}(\sigma'_2.h)
\] (118)

\[
O_2 = O'_2 \land U_2 = U'_2 \land T_2 = T'_2 \land \sigma_2.R = \sigma'_2.R \land \sigma_2.rt = \sigma'_2.rt
\] (119)

\[
\forall x, t \in T_2. \sigma_2.s(x, t) = \sigma'_2.s(x, t)
\] (120)

\[
\forall tid, o. \text{ root } tid \in O(o) \rightarrow o \in \text{dom}(h)
\] (121)

\[
\forall tid, o. \text{ root } tid \in O(o) \rightarrow o \in \text{dom}(h')
\] (122)

To prove all relations (78-83) we assume (114) which is to assume \(T\) as subset of reader threads. Let \(\sigma'_2\) be \(\sigma_2\), \(F'_2\) be \(F_2\). \(O_2\) need not change so we pick \(O'_2\) as \(O_2\). Since \(T_2\) is subset of reader threads, we pick \(T_2\) as \(T'_2\). By assuming (102) we show (111). (117) and (118) follow trivially. (120) follows from choice of \(\sigma'_2\) and (114). (115) and (116) follow from (114) and choice of \(F'_2\). (119), (121) and (122) are determined by choice of \(\sigma'_2\), operational semantics and choices made on maps related to the assertions.

(105) follows from assumptions (113) (80) and choice of \(O'_2\) as \(O_2\).

To show (106) we consider two cases: \(\sigma'_1 \cap \sigma'_2 = \emptyset\) and \(\sigma'_1 \cap \sigma'_2 \neq \emptyset\). First is trivial. Second follows from (110) OW-HD and (111) OW-HD. (107), (109) and (108) are trivial by choices on related maps and semantics of the composition operators for these maps. All compositions shown let us to derive conclusion for \((\sigma'_1, O'_1, U'_1, T'_1, F'_1) \bullet (\sigma'_2, O'_2, U'_2, T'_2, F'_2)\).

**Lemma 16 (ReadStack).**

\[
[z := x]([[\Gamma, z : \_ \_ x : \text{ rcultr } \rho \ 
N]_{M,tid \ast \{m\}}]) \subseteq
[[\Gamma, x : \text{ rcultr } \rho \ 
N, z : \text{ rcultr } \rho \ 
N] \ast R(\{m\})]
\]

**Proof.** We assume

\[
(\sigma, O, U, T, F) \in [[\Gamma, z : \_ \_ x : \text{ rcultr } \rho \ 
N]_{M,tid \ast \{m\}}]
\] (123)

\[
\text{WellFormed}(\sigma, O, U, T, F)
\] (124)

From the assumption in the type rule of T-ReadS we assume that

\[
\text{FV}(\Gamma) \cap \{z\} = \emptyset
\] (125)
We split the composition in [123] as

\[(\sigma, O_1, U_1, T_1, F_1) \in \llbracket \Gamma, z : \text{rcultr} \ 0 \ N \rrbracket_{M,tid}\]  
(126)

\[\sigma_1 \cdot \sigma_2 = \sigma\]  
(127)

\[O_1 \cdot O_2 = O\]  
(128)

\[U_1 \cup U_2 = U\]  
(129)

\[T_1 \cup T_2 = T\]  
(130)

\[F_1 \uplus F_2 = F\]  
(131)

WellFormed(\(\sigma_1, O_1, U_1, T_1, F_1\))  
(133)

WellFormed(\(\sigma_2, O_2, U_2, T_2, F_2\))  
(134)

We must show \(\exists \sigma'_1, \sigma'_2, O'_1, O'_2, U'_1, U'_2, T'_1, T'_2, F'_1, F'_2\) such that

\[(\sigma'_1, O'_1, U'_1, T'_1, F'_1) \in \llbracket \Gamma, z : \text{rcultr} \ 0 \ N \rrbracket_{M,tid}\]  
(135)

\[(\sigma'_2, O'_2, U'_2, T'_2, F'_2) \in \mathcal{R}(\{m\})\]  
(136)

\[\sigma'_1 \cdot \sigma'_2 = \sigma'\]  
(137)

\[O'_1 \cdot O'_2 = O'\]  
(138)

\[U'_1 \cup U'_2 = U'\]  
(139)

\[T'_1 \cup T'_2 = T'\]  
(140)

\[F'_1 \uplus F'_2 = F'\]  
(141)

WellFormed(\(\sigma'_1, O'_1, U'_1, T'_1, F'_1\))  
(142)

WellFormed(\(\sigma'_2, O'_2, U'_2, T'_2, F'_2\))  
(143)

Let \(s(x, \text{tid})\) be \(o_x\). We also know from operational semantics that the machine state has changed as

\[\sigma' = \sigma[s(z, \text{tid}) \mapsto o_x]\]  
(144)

We know that there exists no change in the observation of heap locations

\[O'_1 = O_1\]  
(145)

146 follows from 123

\[T_1 = \{\text{tid}\} \text{ and } \text{tid} = \sigma.l\]  
(146)

\(\sigma'_1\) is determined by operational semantics. The undefined map, \(T_1\) and free list need not change so we can pick \(U'_1\) as \(U_1\), \(T'_1\) as \(T_1\) and \(F'_1\) as \(F_1\). Assuming 126 and choices on maps makes \((\sigma'_1, O'_1, U'_1, T'_1, F'_1)\) in denotation

\[\llbracket \Gamma, z : \text{rcultr} \ 0 \ N \rrbracket_{M,tid}\]  

In the rest of the proof, we prove 142, 143 and show the composition of (\(\sigma'_1, O'_1, U'_1, T'_1, F'_1\)) and (\(\sigma'_2, O'_2, U'_2, T'_2, F'_2\)). 142 follows from 133 trivially.
To prove (138) we need to show interference relation

\[(\sigma, O_2, U_2, T_2, F_2) R (\sigma', O'_2, U'_2, T'_2, F'_2)\]

which by definition means that we must show

\[\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma'_2.h \land \sigma_2.l = \sigma'_2.l)\]

(147)

\[l \in T_2 \rightarrow F_2 = F'_2\]

(148)

\[\forall tid, o. \text{iterator } tid \in O_2(o) \rightarrow o \in \text{dom}(\sigma_2.h)\]

(149)

\[\forall tid, o. \text{iterator } tid \in O_2(o) \rightarrow o \in \text{dom}(\sigma'_2.h)\]

(150)

\[O_2 = O'_2 \land U_2 = U'_2 \land T_2 = T'_2 \land \sigma_2.R = \sigma'_2.R \land \sigma_2.rt = \sigma'_2.rt\]

(151)

\[\forall x, t \in T_2. \sigma_2.s(x, t) = \sigma'_2.s(x, t)\]

(152)

\[\forall tid, o. \text{root } tid \in O(o) \rightarrow o \in \text{dom}(h)\]

(153)

\[\forall tid, o. \text{root } tid \in O(o) \rightarrow o \in \text{dom}(h')\]

(154)

To prove all relations (147-152) we assume (139) which is to assume \(T_2\) as subset of reader threads. Let \(\sigma'_2\) be \(\sigma_2\). \(O_2\) need not change so we pick \(O'_2\) as \(O_2\). We pick \(F'_2\) as \(F_2\). Since \(T_2\) is subset of reader threads, we pick \(T_2\) as \(T'_2\). By assuming (134) we show (143). (150), (153), (154) follow trivially. (152) follows from choice of \(\sigma'_2\) and (141) determined by operational semantics). (147) and (148) follow from (146) and choice of \(F'_2\). (151), (153) and (154) are determined by choice of \(\sigma'_2\), operational semantics and choices made on maps related to the assertions.

(138-141) are trivial by choices on related maps and semantics of the composition operators for these maps. (137) follows trivially from (128). All compositions shown let us to derive conclusion for \((\sigma'_1, O'_1, U'_1, T'_1, F'_1) \circ (\sigma'_2, O'_2, U'_2, T'_2, F'_2)\) trivial.

Lemma 17 (ReadHeap).

\[[z := x.f]\left(\left[[\Gamma, z : x : \text{rcultr} \rho N]\|_{M,tid} \ast \{m\}\right]\right) \subseteq [\left[[\Gamma, x : \text{rcultr} \rho N[f \mapsto z], z : \text{rcultr} \rho' N_0]\|_{M,tid} \ast \{m\}\right]]\]

Proof. We assume

\[(\sigma, O, U, T, F) \in [\left[[\Gamma, z : \text{rcultr} \rho N]\|_{M,tid} \ast \{m\}\right]\]

WellFormed(\sigma, O, U, T, F)

(155)

(156)

From the assumption in the type rule of T-READH we assume that

\[\text{FV}(\Gamma) \cap \{z\} = \emptyset\]

\[\rho.f = \rho'\]

(157)

(158)

(159)
We split the composition in (155) as

\[(\sigma_1, O_1, U_1, T_1, F_1) \in \llbracket \Gamma, x : \text{rcultr } \rho \ N \rrbracket_{M, \text{id}} \] (160)
\[(\sigma_2, O_2, U_2, T_2, F_2) = m \] (161)
\[\sigma_1 \cdot \sigma_2 = \sigma \] (162)
\[O_1 \cdot O_2 = O \] (163)
\[U_1 \cup U_2 = U \] (164)
\[T_1 \cup T_2 = T \] (165)
\[F_1 \cup F_2 = F \] (166)
WellFormed(\sigma_1, O_1, U_1, T_1) (167)
WellFormed(\sigma_2, O_2, U_2, T_2) (168)

We must show \(\exists \sigma'_1, \sigma'_2, O'_1, O'_2, U'_1, U'_2, T'_1, T'_2, F'_1, F'_2\) such that

\[(\sigma'_1, O'_1, U'_1, T'_1, F_1) \in \llbracket \Gamma, x : \text{rcultr } \rho \ N'[f \mapsto z], z : \text{rcultr } \rho' \ N_0 \rrbracket_{M, \text{id}} \] (169)
\[N'(f) = z \] (170)
\[(\sigma'_2, O'_2, U'_2, T'_2, F_2) \in \mathcal{R}(\{m\}) \] (171)
\[\sigma'_1 \cdot \sigma'_2 = \sigma' \] (172)
\[O'_1 \cdot O'_2 = O' \] (173)
\[U'_1 \cup U'_2 = U' \] (174)
\[T'_1 \cup T'_2 = T' \] (175)
\[F'_1 \cup F'_2 = F' \] (176)
WellFormed(\sigma'_1, O'_1, U'_1, T'_1, F'_1) (177)
WellFormed(\sigma'_2, O'_2, U'_2, T'_2, F'_2) (178)

Let \(h(s(z, \text{tid}), f)\) be \(o_2\). We also know from operational semantics that the machine state has changed as

\[\sigma'_1 = \sigma_1[s(x, \text{tid}) \mapsto o_2] \] (179)

\(170\) is determined directly from operational semantics.

We know that there exists no change in the observation of heap locations

\[O'_1 = O_1 \] (180)

\(181\) follows from \(155\)

\[T_1 = \{\text{tid}\} \] and \(\text{tid} = \sigma.l \) (181)

\(\sigma'_1\) is determined by operational semantics. The undefined map, free list and \(T_1\) need not change so we can pick \(U'_1\) as \(U_1\), \(F'_1\) as \(F_1\) and \(T'_1\) and \(T_1\). Assuming \(160\) and choices on maps makes \((\sigma'_1, O'_1, U'_1, T'_1, F'_1)\) in denotation

\[\llbracket \Gamma, x : \text{rcultr } \rho \ N'[f \mapsto z], z : \text{rcultr } \rho' \ N_0 \rrbracket_{M, \text{id}} \]
In the rest of the proof, we prove 177, 178 and show the composition of 
\((\sigma'_1, O'_1, U'_1, T'_1, F'_1)\) and \((\sigma'_2, O'_2, U'_2, T'_2, F'_2)\).

To prove 171 we need to show that each of the memory axioms in Section A.2 holds for the state \((\sigma', O'_1, U'_1, T'_1, F'_1)\).

**Case 31.** - UNQR By 179, 167.

**Case 32.** - OW By 179, 180 and 167.

**Case 33.** - R WOW By 179, 180 and 167.

**Case 34.** - A WR T Trivial.

**Case 35.** - IFL By 169, 167.

**Case 36.** - FLR By operational semantics 169, choice for \(F'_1\) and 167.

**Case 37.** - WULK By 167 WULK, 180 and operational semantics\((\sigma.l = \sigma.l')\).

**Case 38.** - WF, FNR, FPI and FR Trivial.

**Case 39.** - HD

**Case 40.** - WNR By 181 and operational semantics\((\sigma.l = \sigma.l')\).

**Case 41.** - RINFL By operational semantics 179 bounding threads have not changed. We choose \(F'_1\) as \(F_1\). These two together with 167 shows RINFL.

**Case 42.** - ULKR Trivial.

**Case 43.** - UNQRT By 167 UNQRT, 180 and 179.

To prove 171 we need to show interference relation

\((\sigma, O_2, U_2, T_2, F_2) R (\sigma', O'_2, U'_2, T'_2, F'_2)\)

which by definition means that we must show

\[
\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma'_2.h \land \sigma_2.l = \sigma'_2.l) \tag{182}
\]

\[
l \in T_2 \rightarrow F_2 = F'_2 \tag{183}
\]

\[
\forall tid, o. \text{ iterator } tid \in O_2(o) \rightarrow o \in \text{ dom}(\sigma_2.h) \tag{184}
\]

\[
\forall tid, o. \text{ iterator } tid \in O_2(o) \rightarrow o \in \text{ dom}(\sigma'_2.h) \tag{185}
\]

\[
O_2 = O'_2 \land U_2 = U'_2 \land T_2 = T'_2 \land \sigma_2.R_2 = \sigma'_2.R_2 \land \sigma_2.rt = \sigma'_2.rt \tag{186}
\]

\[
\forall x, t \in T_2, \sigma_2.s(x, t) = \sigma'_2.s(x, t) \tag{187}
\]

\[
\forall tid, o. \text{ root } tid \in O(o) \rightarrow o \in \text{ dom}(h) \tag{188}
\]

\[
\forall tid, o. \text{ root } tid \in O(o) \rightarrow o \in \text{ dom}(h') \tag{189}
\]

To prove all relations 182-187 we assume 131 which is to assume \(T_2\) as subset of reader threads. Let \(\sigma'_2\) be \(\sigma_2\) and \(F'_2\) be \(F_2\). \(O_2\) need not change so we pick
$O'_2$ as $O_2$. Since $T_2$ is subset of reader threads, we pick $T_2$ as $T'_2$. By assuming [168] we show [178] and [183] follows trivially. [187] follows from choice of $\sigma'_2$ and [179] determined by operational semantics).

[182] and [185] follow from [181] and choice of $F'_2$. [188] and [189] are determined by choice of $\sigma'_2$, operational semantics and choices made on maps related to the assertions. [173-176] are trivial by choices on related maps and semantics of the composition operators for these maps. [172] follows trivially from [162]. All compositions shown let us to derive conclusion for $(\sigma'_1, O'_1, U'_1, T'_1, F'_1) \bullet (\sigma'_2, O'_2, U'_2, T'_2, F'_2)$.

**Lemma 18 (WriteFreshField).**

\[
[p.f := z][[\Gamma, p : rcuFresh N'_{f,g}, x : rcultr \rho N]_{M,tid} * \{m\}] \subseteq
[[\Gamma, p : rcuFresh N'(\cup f \mapsto z), x : rcultr \rho N([f \mapsto z]) * R(\{m\})]
\]

**Proof.** We assume

\[(\sigma, O, U, T, F) \in \llbracket \Gamma, p : rcuFresh N'_{f,g}, x : rcultr \rho N]_{M,tid} * \{m\} \]

WellFormed$(\sigma, O, U, T, F)$ (191)

From the assumption in the type rule of T-WriteFH we assume that

\[z : rcultr \rho f, z = z \text{ and } f \notin dom(N')\] (192)

We split the composition in [190] as

\[(\sigma, O_1, U_1, T_1, F_1) \in \llbracket \Gamma, p : rcuFresh N'_{f,g}, x : rcultr \rho N]_{M,tid} \]

$(\sigma, O_2, U_2, T_2, F_2) = m$ (193)

$\sigma_1 \cdot \sigma_2 = \sigma$ (194)

$O_1 \cdot O_2 = O$ (195)

$U_1 \cup U_2 = U$ (196)

$T_1 \cup T_2 = T$ (197)

$F_1 \cup F_2 = F$ (198)

WellFormed$(\sigma_1, O_1, U_1, T_1, F_1)$ (199)

WellFormed$(\sigma_2, O_2, U_2, T_2, F_2)$ (200)
We must show \( \exists \sigma', \sigma', O_1', U_1', T_1', F_1' \) such that
\[
(\sigma', O_1', U_1', T_1', F_1') \in \llbracket \Gamma, p : \text{rcuFresh } N(\cup f \rightarrow z), x : \text{rcultr } \rho N([f \rightarrow z]) \rrbracket_{M, tid}
\]
\[N(f) = z \land N'(f) = z \quad (203)\]
\[(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \quad (204)\]
\[\sigma_1' \cdot \sigma_2' = \sigma' \quad (205)\]
\[O_1' \cdot O_2' = O' \quad (206)\]
\[U_1' \cup U_2' = U' \quad (207)\]
\[T_1' \cup T_2' = T' \quad (208)\]
\[F_1' \sqcup F_2' = F' \quad (209)\]
\[\text{WellFormed}(\sigma'_1, O'_1, U'_1, T'_1, F'_1) \quad (210)\]
\[\text{WellFormed}(\sigma'_2, O'_2, U'_2, T'_2, F'_2) \quad (211)\]

We also know from operational semantics that the machine state has changed as
\[
\sigma' = \sigma[h(s(p, tid), f) \mapsto s(z, tid)] \quad (212)\]

There exists no change in the observation of heap locations
\[
O_1' = O_1 \quad (213)\]

\[214\text{ follows from } 199\]
\[T_1 = \{tid\} \text{ and } tid = \sigma.l \quad (214)\]

\(\sigma'_1\) is determined by operational semantics. The undefined map, free list, \(T_1\) need not change so we can pick \(U'_1\) as \(U_1\), \(T'_1\) as \(T_1\) and \(F'_1\) as \(F_1\). Assuming \(193\) and choices on maps makes \((\sigma'_1, O'_1, U'_1, T'_1, F'_1)\) in denotation
\[
\llbracket \Gamma, p : \text{rcuFresh } N(\cup f \rightarrow z), x : \text{rcultr } \rho N([f \rightarrow z]) \rrbracket_{M, tid}
\]

In the rest of the proof, we prove \(210\) and \(211\) and show the composition of
\((\sigma'_1, O'_1, U'_1, T'_1, F'_1)\) and \((\sigma'_2, O'_2, U'_2, T'_2, F'_2)\). To prove \(210\), we need to show that each of the memory axioms in Section A.2 holds for the state \((\sigma', O'_1, U'_1, T'_1, F'_1)\).

**Case 44. - UNQR** By \(200\) UNQR, \(210\) PR (proved) and \(\sigma.rt = \sigma.rt'\).

**Case 45. - OW** By \(212, 213\) and \(200\) OW

**Case 46. - RWOW** By \(212, 213\) and \(200\) RWOW

**Case 47. - AWRT** Trivial.

**Case 48. - IFL** By \(200\) WULK, \(213\) choice of \(F'_1\) and operational semantics.
Case 49. - FLR By operational semantics [212], choice of $F'_1$ and [200].

Case 50. - WULK By [200], WULK, [213] and operational semantics($\sigma.l = \sigma.l'$).

Case 51. - WF By [200], WF, [214] and operational semantics [214].

Case 52. - FR By [200], FR, [214] and operational semantics [214].

Case 53. - FNR By [200], FNR, [214] and operational semantics [214].

Case 54. - FPI By [200], FPI, [193] and [192].

Case 55. - HD

Case 56. - WNR By [214] and operational semantics($\sigma.l = \sigma.l'$).

Case 57. - RINFL By operational semantics [212] - bounding threads have not changed), choice of $F'_1$ and [200].

Case 58. - ULKR Trivial.

Case 59. - UNQRT By [200], UNQRT, [213] and [212].

To prove [204] we need to show interference relation

$$(\sigma, O_2, U_2, T_2, F_2)R(\sigma', O'_2, U'_2, T'_2, F'_2)$$

which by definition means that we must show

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma'_2.h \land \sigma_2.l = \sigma'_2.l)$$

(215)

$$l \in T_2 \rightarrow F_2 = F'_2$$

(216)

$$\forall tid, o. \text{ iterator } tid \in O_2(o) \rightarrow o \in \text{dom}(\sigma_2.h)$$

(217)

$$\forall tid, o. \text{ iterator } tid \in O_2(o) \rightarrow o \in \text{dom}(\sigma'_2.h)$$

(218)

$$O_2 = O'_2 \cup U_2 = U'_2 \cup T_2 = T'_2 \cup \sigma_2.R = \sigma'_2.R \land \sigma_2.rt = \sigma'_2.rt$$

(219)

$$\forall x, t \in T_2, \sigma_2.s(x, t) = \sigma'_2.s(x, t)$$

(220)

$$\forall tid, o. \text{ root } tid \in O(o) \rightarrow o \in \text{dom}(h)$$

(221)

$$\forall tid, o. \text{ root } tid \in O(o) \rightarrow o \in \text{dom}(h')$$

(222)

To prove all relations [215-220] we assume [214] which is to assume $T_2$ as subset of reader threads and [201]. Let $\sigma'_2$ be $\sigma_2$ and $F'_2$ be $F_2$. $O_2$ need not change so we pick $O'_2$ as $O_2$. Since $T_2$ is subset of reader threads, we pick $T_2$ as $T'_2$. By assuming [201] we show [211] and [217] follows trivially. [220] follows from choice of $\sigma'_2$ and [212] (determined by operational semantics).

[215] and [216] follow from [214] and choice of $F'_2$, [219] are determined by operational semantics, choice of $\sigma'_2$ and choices made on maps related to the assertions.

[207]-[209] are trivial by choices on related maps and semantics of the composition operators for these maps. [221] and [222] follow from choice of $\sigma'_2$.

$O'_1 \bullet O'_2$ follows from [196], [213] and choice of $O_2$.

We assume $\sigma_1.h \bullet \sigma_2.h$. We know from [172] that $f \notin \text{dom}(N')$. From [202], [210], [211], FNR, [210], [211], RTR and [210],[211], WNR we show $\sigma'_1.h \bullet \sigma'_2.h$ (with choices for other maps in the machine state we show [172]). All compositions shown let us to derive conclusion for $(\sigma'_1, O'_1, U'_1, T'_1, F'_1) \bullet (\sigma'_2, O'_2, U'_2, T'_2, F'_2)$.
Lemma 19 (Sync).

\[
\llbracket \text{SyncStart; SyncStop}\rrbracket \llbracket [\Gamma]_{M,tid} \ast \{m\} \rrbracket \subseteq
\llbracket [\Gamma[x : \text{freeable}] / x : \text{unlinked}] \ast R(\{m\}) \rrbracket
\]

Proof. We assume

\[(\sigma, O, U, T, F) \in [\Gamma]_{M,tid} \ast \{m\} \] (223)
WellFormed$(\sigma, O, U, T, F)$ (224)

We split the composition in (223) as

\[(\sigma, O_1, U_1, T_1, F_1) \in [\Gamma]_{M,tid} \] (225)
\[(\sigma, O_2, U_2, T_2, F_2) = m \] (226)
\[\sigma_1 \cdot \sigma_2 = \sigma \] (227)
\[O_1 \cdot O_2 = O \] (228)
\[U_1 \cup U_2 = U \] (229)
\[T_1 \cup T_2 = T \] (230)
\[F_1 \uplus F_2 = F \] (231)
WellFormed$(\sigma, O_1, U_1, T_1, F_1)$ (232)
WellFormed$(\sigma, O_2, U_2, T_2, F_2)$ (233)

We must show \( \exists \sigma'_1, \sigma'_2, O'_1, O'_2, U'_1, U'_2, T'_1, T'_2, F'_1, F'_2 \) such that

\[(\sigma'_1, O'_1, U'_1, T'_1, F'_1) \in [\Gamma[x : \text{freeable}] / x : \text{unlinked}]_{M,tid} \] (234)
\[(\sigma'_2, O'_2, U'_2, T'_2, F'_2) \in R(\{m\}) \] (235)
\[\sigma'_1 \cdot \sigma'_2 = \sigma' \] (236)
\[O'_1 \cdot O'_2 = O' \] (237)
\[U'_1 \cup U'_2 = U' \] (238)
\[T'_1 \cup T'_2 = T' \] (239)
\[F'_1 \uplus F'_2 = F' \] (240)
WellFormed$(\sigma', O'_1, U'_1, T'_1, F'_1)$ (241)
WellFormed$(\sigma', O'_2, U'_2, T'_2, F'_2)$ (242)

We also know from operational semantics that SyncStart changes

\[\sigma'_1.B = \sigma_1.B[\emptyset \mapsto R] \] (244)

Then SyncStop changes it to $\emptyset$ so there exists no change in $B$ after SyncStart; SyncStop.
So there is no change in machine state.

\[\sigma'_1 = \sigma_1 \] (245)
There exists no change in the observation of heap locations
\[ O'_1 = O_1 (\forall x \in R \cdot s(x, tid)) \text{ [unlinked \(\mapsto\) freeable]} \] (246)
and we pick free list to be
\[ F'_1 = F_1 (\forall x \cdot \text{unlinked} \in R \cdot s(x, tid) [T \mapsto \{\}] ) \] (247)

(248) follows from (228)

\[ T_1 = \{tid\} \text{ and } tid = \sigma.l \] (248)

Let \( T'_1 \) be \( T_1 \) and \( \sigma'_1 \) (not changed) be determined by operational semantics. The undefined map need not change so we can pick \( U'_1 \) as \( U_1 \). Assuming (227) and choices on maps makes \( (\sigma'_1, O'_1, U'_1, T'_1, F'_1) \) in denotation
\[ \llbracket P[x: \text{freeable} / x: \text{unlinked}] \rrbracket_{M, tid} \]

In the rest of the proof, we prove (242) and (243) and show the composition of \( (\sigma'_1, O'_1, U'_1, T'_1, F'_1) \) and \( (\sigma'_2, O'_2, U'_2, T'_2, F'_2) \). To prove (242) we need to show that each of the memory axioms in Section A.2 holds for the state \( (\sigma', O'_1, U'_1, T'_1, F'_1) \) which is trivial by assuming (242). We also know (254) as we showed the support of state to the denotation).

To prove (233) we need to show interference relation
\( (\sigma, O_2, U_2, T_2, F_2) R (\sigma', O'_2, U'_2, T'_2, F'_2) \)
which by definition means that we must show
\[ \sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma'_2.h \land \sigma_2.l = \sigma'_2.l) \] (249)
\[ l \in T_2 \rightarrow F_2 = F'_2 \] (250)
\[ \forall tid, o. \text{iterator } tid \in O_2(o) \rightarrow o \in \text{dom}(\sigma_2.h) \] (251)
\[ \forall tid, o. \text{iterator } tid \in O_2(o) \rightarrow o \in \text{dom}(\sigma'_2.h) \] (252)
\[ O_2 = O'_2 \land U_2 = U'_2 \land T_2 = T'_2 \land \sigma_2.R = \sigma'_2.R \land \sigma_2.rt = \sigma'_2.rt \] (253)
\[ \forall x, t \in T_2, \sigma_2.s(x, t) = \sigma'_2.s(x, t) \] (254)
\[ \forall tid, o. \text{root } tid \in O(o) \rightarrow o \in \text{dom}(h) \] (255)
\[ \forall tid, o. \text{root } tid \in O(o) \rightarrow o \in \text{dom}(h') \] (256)

To prove all relations (249)-(251) we assume (218) which is to assume \( T_2 \) as subset of reader threads and (263). Let \( \sigma'_2 \) be \( \sigma_2 \). \( O_2 \) need not change so we pick \( O'_2 \) as \( O_2 \). Since \( T_2 \) is subset of reader threads, we pick \( T_2 \) as \( T'_2 \). By assuming (233) we show (243) (251) and (252) follows trivially. (251) follows from choice of \( \sigma'_2 \) and (247) (determined by operational semantics).

(249) and (250) follow from (248) (253) are determined by choice of \( \sigma'_2 \) and operational semantics and choices made on maps related to the assertions.

(235) and (241) follow from (229) (241) (243) (247) and choice of \( \sigma'_2 \). All compositions shown let us to derive conclusion for \( (\sigma'_1, O'_1, U'_1, T'_1, F'_1) \bullet (\sigma'_2, O'_2, U'_2, T'_2, F'_2) \).
Lemma 20 (Alloc).

\[
\llbracket x := \text{new}\rrbracket(\llbracket \Gamma, x : \text{undef}\rrbracket_{M,tid} * \{m\}) \subseteq \\
\llbracket \Gamma, x : \text{rcuFresh}_N \rrbracket \ast \mathcal{R}(\{m\})
\]

Proof. We assume

\[
(\sigma, O, U, T, F) \in \llbracket \Gamma, x : \text{undef}\rrbracket_{M,tid} \text{ (257)}
\]

WellFormed(\sigma, O, U, T, F) \text{ (258)}

We split the composition in 257 as

\[
(\sigma, O_1, U_1, T_1, F_1) \in \llbracket \Gamma, x : \text{undef}\rrbracket_{M,tid} \text{ (259)}
\]

\[
(\sigma, O_2, U_2, T_2, F_2) = m \text{ (260)}
\]

\[
\sigma_1 \cdot \sigma_2 = \sigma \text{ (261)}
\]

\[
O_1 \cdot O_2 = O \text{ (262)}
\]

\[
U_1 \cup U_2 = U \text{ (263)}
\]

\[
T_1 \cup T_2 = T \text{ (264)}
\]

\[
F_1 \uplus F_2 = F \text{ (265)}
\]

WellFormed(\sigma, O_1, U_1, T_1, F_1) \text{ (266)}

WellFormed(\sigma, O_2, U_2, T_2, F_2) \text{ (267)}

We must show \exists O'_1, O'_2, U'_1, U'_2, T'_1, T'_2, F'_1, F'_2 such that

\[
(\sigma', O'_1, U'_1, T'_1, F'_1) \in \llbracket \Gamma, x : \text{rcuFresh}_N \rrbracket \text{ (268)}
\]

\[
(\sigma', O'_2, U'_2, T'_2, F'_2) \in \mathcal{R}(\{m\}) \text{ (269)}
\]

\[
\sigma'_1 \cdot \sigma'_2 = \sigma' \text{ (270)}
\]

\[
O'_1 \cdot O'_2 = O' \text{ (271)}
\]

\[
U'_1 \cup U'_2 = U' \text{ (272)}
\]

\[
T'_1 \cup T'_2 = T' \text{ (273)}
\]

\[
F'_1 \uplus F'_2 = F' \text{ (274)}
\]

WellFormed(\sigma', O'_1, U'_1, T'_1) \text{ (275)}

WellFormed(\sigma', O'_2, U'_2, T'_2) \text{ (276)}

From operational semantics we know that \( s(y, \text{tid}) = \ell \). We also know from operational semantics that the machine state has changed as

\[
\sigma' = \sigma[h(\ell) \mapsto \text{nullmap}] \text{ (277)}
\]

There exists no change in the observation of heap locations

\[
O'_1 = O_1(\ell)[\text{undef} \mapsto \text{fresh}] \text{ (278)}
\]
$T_1 = \{\text{tid}\}$ and $\text{tid} = \sigma.l$ (279)

Let $T'_1$ to be $T_1$. Undefined map and free list need not change so we can pick $U'_1$ as $U_1$ and $F'_1$ as $F_1$ and show (268) that $(\sigma', O'_1, U'_1, T'_1, F'_1)$ is in denotation of $\llbracket F, x : \text{rcuFresh}, N \rrbracket$

In the rest of the proof, we prove (275) and $(\sigma'_1, O'_1, U'_1, T'_1, F'_1)$ and $(\sigma'_2, O'_2, U'_2, T'_2, F'_2)$. To prove (275) we need to show that each of the memory axioms in Section A.2 holds for the state $(\sigma'_1, O'_1, U'_1, T'_1, F'_1)$.

**Case 60.** - UNQR Determined by (268) and operational semantics ($\ell$ is fresh-unique).

**Case 61.** - RWOW, OW By (268)

**Case 62.** - AWRT Determined by operational semantics ($\ell$ is fresh-unique).

**Case 63.** - IFL, ULKR, WULK, RINFL, UNQRT Trivial.

**Case 64.** - FLR determined by operational semantics and (268)

**Case 65.** - WF By (270) and (278)

**Case 66.** - FR Determined by operational semantics ($\ell$ is fresh-unique).

**Case 67.** - FNR By (268) and operational semantics ($\ell$ is fresh-unique).

**Case 68.** - FPI By (268) and $N_{f,g}$.

**Case 69.** - HD

**Case 70.** - WNR By (270)

To prove (269) we need to show interference relation

$$(\sigma, O_2, U_2, T_2, F_2)R(\sigma', O'_2, U'_2, T'_2, F'_2)$$

which by definition means that we must show

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma'_2.h \land \sigma_2.l = \sigma'_2.l)$$

$$l \in T_2 \rightarrow F_2 = F'_2$$

$$\forall \text{tid}, o. \text{iterator \ tid} \in O_2(o) \rightarrow o \in \text{dom} (\sigma_2.h)$$

$$\forall \text{tid}, o. \text{iterator \ tid} \in O_2(o) \rightarrow o \in \text{dom} (\sigma'_2.h)$$

$$O_2 = O'_2 \land U_2 = U'_2 \land T_2 = T'_2 \land \sigma_2.R = \sigma'_2.R \land \sigma_2.rt = \sigma'_2.rt$$

$$\forall x, t \in T. \sigma_2.s(x, t) = \sigma'_2.s(x, t)$$

$$\forall \text{tid}, o. \text{root \ tid} \in O(o) \rightarrow o \in \text{dom} (h)$$

$$\forall \text{tid}, o. \text{root \ tid} \in O(o) \rightarrow o \in \text{dom} (h')$$

(280) (281) (282) (283) (284) (285) (286) (287)
To prove all relations (280-287) we assume (279) which is to assume \( T_2 \) as subset of reader threads and (280). Let \( \sigma'_2 \) be \( \sigma_2 \), \( F_2 \) and \( O_2 \) need not change so we pick \( O'_2 \) as \( O_2 \) and \( F'_2 \) as \( F_2 \). Since \( T_2 \) is subset of reader threads, we pick \( T_2 \) as \( T'_2 \). By assuming (279) and choices on maps we show (280). (282) and (283) follow trivially. (284) follows from choice of \( \sigma'_2 \) and (277) determined by operational semantics. (274) follow from (262)-(273) semantics of compositions operators and choices made for maps of the logical state.

(280) and (281) follow from (279) and choice on \( F'_2 \). (284) are determined by operational semantics, operational semantics and choices made on maps related to the assertion.

\( \sigma'_1.h \cap \sigma'_2.h = \emptyset \) is determined by operational semantics (\( \ell \) is unique and fresh). So, (280) follows from (261) and choice of \( \sigma'_2 \). All compositions shown let us to derive conclusion for \((\sigma'_1, O'_1, U'_1, T'_1, F'_1) \bullet (\sigma'_2, O'_2, U'_2, T'_2, F'_2)\).

**Lemma 21 (Free).**

\[
\llbracket \text{Free}(x) \rrbracket (\llbracket x : \text{freeable}\rrbracket_{M,tid} \ast \{m\}) \subseteq \llbracket x : \text{undef} \rrbracket \ast R(\{m\})
\]

**Proof.** We assume

\[
(\sigma, O, U, T, F) \in \llbracket x : \text{freeable}\rrbracket_{M,tid} \ast \{m\}) \quad (288)
\]

WellFormed(\(\sigma, O, U, T, F\)) \quad (289)

We split the composition in (288) as

\[
(\sigma, O_1, U_1, T_1, F_1) \in \llbracket x : \text{freeable}\rrbracket_{M,tid} \quad (290)
\]

\( (\sigma, O_2, U_2, T_2, F_2) = m \) \quad (291)

\( \sigma_1 \bullet \sigma_2 = \sigma \) \quad (292)

\( O_1 \bullet O_2 = O \) \quad (293)

\( U_1 \cup U_2 = U \) \quad (294)

\( T_1 \cup T_2 = T \) \quad (295)

\( F_1 \uplus F_2 = F \) \quad (296)

WellFormed(\(\sigma, O_1, U_1, T_1, F_1\)) \quad (297)

WellFormed(\(\sigma, O_2, U_2, T_2, F_2\)) \quad (298)
We must show \( \exists O_1', O_2', U_1', U_2', T_1', T_2', F_1', F_2' \) such that

\[
\begin{align*}
& (\sigma', O_1', U_1', T_1', F_1') \in [x : \text{undef}] \\
& (\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \\
& \sigma_1 \bullet \sigma_2 = \sigma' \\
& O_1' \bullet O_2' = O' \\
& U_1' \cup U_2' = U' \\
& T_1' \cup T_2' = T' \\
& F_1' \uplus F_2' = F' \\
& \text{WellFormed}(\sigma', O_1', U_1', T_1', F_1') \\
& \text{WellFormed}(\sigma', O_2', U_2', T_2', F_2')
\end{align*}
\] (299)

(300)

(301)

(302)

(303)

(304)

(305)

(306)

(307)

From operational semantics we know that

\[
\forall f, o' . \text{rt} \neq s(x, \text{tid}) \land o' \neq s(x, \text{tid}) \implies h(o', f) = h'(o', f) \land \forall f . h'(o, f) = \text{undef}
\] (308)

\[
O_1' = O_1(s(x, \text{tid}))[\text{freeable} \mapsto \text{undef}]
\] (309)

\[
F_1' = F_1 \setminus \{s(x, \text{tid}) \mapsto \emptyset\}
\] (310)

\[
U_1' = U_1 \cup \{(x, \text{tid})\}
\] (311)

(312) follows from (288)

\[
T_1 = \{\text{tid}\} \text{ and } \text{tid} = \sigma.l
\] (312)

Let \( T_1' \) to be \( T_1 \). All (308-311) show (299) that \((\sigma', O_1', U_1', T_1', F_1')\) is in denotation of

\[
[x : \text{undef}]
\]

In the rest of the proof, we prove (308-311) and show the composition of \((\sigma_1', O_1', U_1', T_1', F_1')\) and \((\sigma_2', O_2', U_2', T_2', F_2')\). To prove (308), we need to show that each of the memory axioms in Section A.2 holds for the state \((\sigma', O_1', U_1', T_1', F_1')\) and it trivial by (308-311) and (299).

To prove (308) we need to show interference relation

\[
(\sigma, O_2, U_2, T_2, F_2) \mathcal{R} (\sigma', O_2', U_2', T_2', F_2')
\]
which by definition means that we must show

\[ \sigma_2, l \in T_2 \rightarrow (\sigma_2, h = \sigma_2', h \land \sigma_2, l = \sigma_2', l) \]  
(313)

\[ l \in T_2 \rightarrow F_2 = F'_2 \]  
(314)

\[ \forall \text{tid}, o. \text{iterator} \text{tid} \in O_2(o) \rightarrow o \in dom(\sigma_2, h) \]  
(315)

\[ \forall \text{tid}, o. \text{iterator} \text{tid} \in O_2(o) \rightarrow o \in dom(\sigma_2', h) \]  
(316)

\[ O_2 = O_2' \land U_2 = U_2' \land T_2 = T_2' \land \sigma_2, R = \sigma_2', R \land \sigma_2, rt = \sigma_2', rt \]  
(317)

\[ \forall x, t \in T. \sigma_2, s(x, t) = \sigma_2', s(x, t) \]  
(318)

\[ \forall \text{root} \text{tid} \in O(o) \rightarrow o \in dom(h) \]  
(319)

\[ \forall \text{root} \text{tid} \in O(o) \rightarrow o \in dom(h') \]  
(320)

To prove all relations (313-320) we assume (312) which is to assume \( T_2 \) as subset of reader threads and (267). Let \( \sigma_2' \) be \( \sigma_2 \). \( F_2 \) and \( O_2 \) need not change so we pick \( O_2' \) as \( O_2 \) and \( F_2' \) as \( F_2 \). Since \( T_2 \) is subset of reader threads, we pick \( T_2' \) as \( T_2' \). By assuming (267) and choices on maps we show (307) (313) and (316) follow trivially. (318) follows from choice of \( \sigma_2' \) and (308) determined by operational semantics). (302) (305) follow from (291) (296) semantics of composition operators and choices on related maps.

(313) and (314) follow from (312) and choice on \( F_2' \). (317) are determined by operational semantics, choice of \( \sigma_2' \) and choices made on maps related to the assertion.

Composition for heap for case \( \sigma_1', h \cap \sigma_2', h = \emptyset \) is trivial. \( \sigma_1', h \cap \sigma_2', h \neq \emptyset \) is determined by semantics of heap composition operator \( \bullet^h \) (v has precedence over \( \text{undef} \)) and this makes showing (302) straightforward. Since other machine components do not change(determined by operational semantics), (301) follows from (292) (298) and choice of \( \sigma_2' \). All compositions shown let us to derive conclusion for \( (\sigma, O, U, T, F) \): \( (\sigma_2', O_2', U_2', T_2', F_2') \).

**Lemma 22 (RReadStack).**

\[ [z := x][[[\Gamma, z : \text{rcultr}, x : \text{rcultr}]_{R, \text{tid}} \ast \{m\}]] \subseteq \]

\[ [[[\Gamma, x : \text{rcultr}, z : \text{rcultr}] \ast \mathcal{R}(\{m\})]] \]

**Proof.** We assume

\[ (\sigma, O, U, T, F) \in [[\Gamma, \Gamma, z : \text{rcultr}, x : \text{rcultr}]_{R, \text{tid}} \ast \{m\}] \]  
(321)

\[ \text{WellFormed}(\sigma, O, U, T, F) \]  
(322)
We split the composition in \([\ref{321}]\) as

\[
(\sigma_1, O_1, U_1, T_1, F_1) \in \langle [\Gamma, \Gamma, z : rcultr, x : rcultr]_{R,tid} \rangle
\]

(323)

\[
(\sigma, O_2, T_2, F_2) = m
\]

(324)

\[
O_1 \cdot O_2 = O
\]

(325)

\[
\sigma_1 \cdot \sigma_2 = \sigma
\]

(326)

\[
U_1 \cup U_2 = U
\]

(327)

\[
T_1 \cup T_2 = T
\]

(328)

\[
F_1 \uplus F_2 = F
\]

(329)

\[
\text{WellFormed}(\sigma, O_1, U_1, T_1, F_1)
\]

(330)

\[
\text{WellFormed}(\sigma, O_2, U_2, T_2, F_2)
\]

(331)

We must show \(\exists O_1', O_2', U_1', U_2', T_1', T_2', F_1', F_2'\) such that

\[
(\sigma', O_1', U_1', T_1', F_1') \in \langle [\Gamma, \Gamma, z : rcultr]_{R,tid} \rangle
\]

(332)

\[
(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\})
\]

(333)

\[
O_1' \cdot O_2' = O'
\]

(334)

\[
\sigma_1' \cdot \sigma_2' = \sigma'
\]

(335)

\[
U_1' \cup U_2' = U'
\]

(336)

\[
T_1' \cup T_2' = T'
\]

(337)

\[
F_1' \uplus F_2' = F'
\]

(338)

\[
\text{WellFormed}(\sigma', O_1', U_1', T_1', F_1')
\]

(339)

\[
\text{WellFormed}(\sigma', O_2', U_2', T_2', F_2')
\]

(340)

We also know from operational semantics that the machine state has changed as

\[
\sigma_1' = \sigma_1
\]

(341)

There exists no change in the observation of heap locations

\[
O_1' = O_1
\]

(342)

(343) follows from (321)

\[
T_1 \subseteq R
\]

(343)

Let \(T_1'\) be \(T_1\) and \(\sigma_1'\) be determined by operational semantics as \(\sigma_1\). The undefined map and free list need not change so we can pick \(U_1'\) as \(U_1\) and \(F_1'\) as \(F_1\). Assuming \(323\) and choices on maps makes \((\sigma_1', O_1', U_1', T_1', F_1')\) in denotation

\[
\langle [\Gamma, \Gamma, z : rcultr]_{R,tid} \rangle
\]

In the rest of the proof, we prove \(339, 340\) and show the composition of \((\sigma_1', O_1', U_1', T_1', F_1')\) and \((\sigma_2', O_2', U_2', T_2', F_2')\). To prove \(339\) we need to show that
each of the memory axioms in Section 2 holds for the state \((\sigma', O'_1, U'_1, T'_1, F'_1)\) which is trivial by assuming \(330\) and knowing \(333, 343, 342\) and components of the state determined by operational semantics.

To prove \(340\) we need to show that WellFormedness is preserved under interference relation

\[(\sigma, O_2, U_2, T_2, F_2) \mathcal{R} (\sigma', O'_2, U'_2, T'_2, F'_2)\]

which by definition means that we must show

\[
\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma'_2.h \land \sigma_2.l = \sigma'_2.l) \tag{344}
\]

\[l \in T_2 \rightarrow F_2 = F'_2 \tag{345}\]

\[\forall tid, o. \text{iterator } tid \in O_2(o) \rightarrow o \in \text{dom}(\sigma_2.h) \tag{346}\]

\[\forall tid, o. \text{iterator } tid \in O_2(o) \rightarrow o \in \text{dom}(\sigma'_2.h) \tag{347}\]

\[O_2 = O'_2 \land U_2 = U'_2 \land T_2 = T'_2 \land \sigma_2.B = \sigma'_2.B \land \sigma_2.rt = \sigma'_2.rt \tag{348}\]

\[\forall x, t \in T_2, \sigma_2.s(x, t) = \sigma'_2.s(x, t) \tag{349}\]

\[\forall tid, o. \text{root } tid \in O(o) \rightarrow o \in \text{dom}(h) \tag{350}\]

\[\forall tid, o. \text{root } tid \in O(o) \rightarrow o \in \text{dom}(h') \tag{351}\]

\(\sigma_2, O_2, U_2\) and \(T_2\) need not change so that we choose \(\sigma'_2\) to be \(\sigma'_2\), \(O'_2\) to be \(O_2\), \(U'_2\) to \(U_2\) and \(T'_2\) to be \(T_2\). Let \(F'_2\) be \(F_2\). These choices make proving \(344\), \(345\) trivial and \(346\), \(347\), \(348\), \(349\) follow from assumptions \(325\), \(329\) choices made for related maps and semantics of composition operations. All compositions shown let us derive conclusion for \((\sigma_1, O_1, U_1, T_1, F_1) \bullet (\sigma'_2, O'_2, U'_2, T'_2, F'_2)\).

Lemma 23 (RReadHeap).

\[
[z := f](\llbracket \Gamma, z : \{rcultr, x : \{rcultr\}_R, tid \} \rrbracket \{m\}) \subseteq \llbracket \Gamma, x : \{rcultr\}_R, z : \{rcultr\}_R \mathcal{R} \{\{m\}\} \rrbracket
\]

Proof. We assume

\[(\sigma, O, U, T, F) \in \llbracket \Gamma, z : \{rcultr\}_R, x : \{rcultr\}_R, tid \} \rrbracket \{m\} \tag{352}\]

WellFormed(\(\sigma, O, U, T, F\)) \(\tag{353}\)

We split the composition in \(352\) as

\[(\sigma_1, O_1, U_1, T_1, F_1) \in \llbracket \Gamma, z : \{rcultr\}_R, x : \{rcultr\}_R, tid \} \rrbracket \tag{354}\]

\[(\sigma_2, O_2, U_2, T_2, F_2) = m \tag{355}\]

\[\sigma_1 \bullet \sigma_2 = \sigma \tag{356}\]

\[O_1 \bullet O_2 = O \tag{357}\]

\[U_1 \cup U_2 = U \tag{358}\]

\[T_1 \cup T_2 = T \tag{359}\]

\[F_1 \uplus F_2 = F \tag{360}\]

WellFormed(\(\sigma_1, O_1, U_1, T_1, F_1\)) \(\tag{361}\)

WellFormed(\(\sigma_2, O_2, U_2, T_2, F_2\)) \(\tag{362}\)
We must show \( \exists \sigma'_1, \sigma'_2, O'_1, O'_2, U'_1, U'_2, T'_1, T'_2, F'_1, F'_2 \) such that

\[
(\sigma'_1, O'_1, U'_1, T'_1, F'_1) \in \llbracket [\Gamma, x: \text{rcultr}, z: \text{rcultr}] \rrbracket
\]

\[
(\sigma'_2, O'_2, U'_2, T'_2, F'_2) \in \mathcal{R}(\{m\})
\]

\[
\sigma'_1 \cdot \sigma'_2 = \sigma'
\]

\[
O'_1 \cdot O'_2 = O'
\]

\[
U'_1 \cup U'_2 = U'
\]

\[
T'_1 \cup T'_2 = T'
\]

WellFormed(\( \sigma'_1, O'_1, U'_1, T'_1, F'_1 \))

WellFormed(\( \sigma'_2, O'_2, U'_2, T'_2, F'_2 \))

Let \( h(s(x, \text{tid}), f) \) be \( o_x \). We also know from operational semantics that the machine state has changed as

\[
\sigma'_1 = \sigma_1[s(z, \text{tid}) \mapsto o_x]
\]

There exists no change in the observation of heap locations

\[
O'_1 = O_1
\]

This follows from 352

\[
T_1 \subseteq R
\]

Proof is similar to Lemma 22.

### A.4 Soundness Proof of Structural Program Actions

In this section, we introduce soundness Theorem 1 for structural rules of the type system. We consider the cases of the induction on derivation of \( \Gamma \vdash C \vdash \Gamma' \) for all type systems, \( R, M \).

Although we have proofs for read-side structural rules, we only present proofs for write-side structural type rules in this section as read-side rules are simple versions of write-side rules and proofs for them are trivial and already captured by proofs for write-side structural rules.

**Theorem 1 (Type System Soundness).**

\( \forall \Gamma, \Gamma', \Delta, \Gamma \vdash C \vdash \Gamma' \implies [\Gamma \vdash C \vdash \Gamma'] \)

**Proof.** Induction on derivation of \( \Gamma \vdash_M C \vdash \Gamma' \).

**Case 71.** -M: consequence where \( C \) has the form \( \Gamma \vdash_M C \vdash \Gamma'' \). We know

\[
\Gamma' \vdash_M C \vdash \Gamma''
\]

\[
\Gamma \vdash \Gamma'
\]

\[
\Gamma'' \vdash \Gamma''
\]

\[
\llbracket [\Delta]_{M, \text{tid}} \rrbracket C\llbracket [\Gamma'']_{M, \text{tid}} \rrbracket
\]
We need to show

\[ \{[\Gamma]_{M,tid}\} C \{[\Gamma''']_{M,tid}\} \]  (378)

The \( \prec \) relation translated to entailment relation in Views Logic. The relation is established over the action judgement for identity label/transition.

From 375 and Lemma 24 we know

\[ [\Gamma]_{M,tid} \sqsubseteq [\Gamma''']_{M,tid} \]  (379)

From 376 and 24 we know

\[ [\Gamma''']_{M,tid} \sqsubseteq [\Gamma''']_{M,tid} \]  (380)

By using 379, 380, and 377 as antecedents of Views Logic’s consequence rule, we conclude 378.

Case 72. -M: where \( C \) is sequence statement. \( C \) has the form \( C_1; C_2 \). Our goal is to prove

\[ \{[\Gamma]_{M,tid}\} \vdash_M C_1; C_2 \vdash \{[\Gamma''']_{M,tid}\} \]  (381)

We know

\[ \Gamma \vdash_M C_1 \vdash \Gamma' \]  (382)
\[ \Gamma' \vdash_M C_2 \vdash \Gamma''' \]  (383)
\[ \{[\Gamma]_{M,tid}\} C_1 \{[\Gamma']_{M,tid}\} \]  (384)
\[ \{[\Gamma']_{M,tid}\} C_2 \{[\Gamma''']_{M,tid}\} \]  (385)

By using 384 and 385 as the antecedents for the Views sequencing rule, we can derive the conclusion for 381.

Case 73. -M: where \( C \) is loop statement. \( C \) has the form while \((x)\) \{\( C \)\}.

\[ \Gamma \vdash_M C \vdash \Gamma \]  (386)
\[ \Gamma(x) = \text{bool} \]  (387)
\[ \{[\Gamma]_{M,tid}\} C \{[\Gamma]_{M,tid}\} \]  (388)

Our goal is to prove

\[ \{[\Gamma]_{M,tid}\} (\text{assume}(x); C)^*; \text{assume}(\neg x) \{[\Gamma]_{M,tid}\} \]  (389)

We prove 389 by from the consequence rule, based on the proofs of the following 390 and 391

\[ \{[\Gamma]_{M,tid}\} (\text{assume}(x); C)^* \{[\Gamma]_{M,tid}\} \]  (390)
The proof of (390) follows from Views Logic’s proof rule for assume construct by using

\[
\{[[\Gamma]]_{M,tid}\} \text{assume} (\neg x) \{[[\Gamma]]_{M,tid}\}
\]  

as antecedent. We can use this antecedent together with the antecedent we know from (388)

\[
\{[[\Gamma]]_{M,tid}\} C \{[[\Gamma]]_{M,tid}\}
\]

as antecedents to the Views Logic’s proof rule for sequencing. Then we use the antecedent

\[
\{[[\Gamma]]_{M,tid}\} \text{assume} (x) \{[[\Gamma]]_{M,tid}\}
\]

to the proof rule for nondeterministic looping.

The proof of (391) follows from Views Logic’s proof rule for assume construct by using the

\[
\{[[\Gamma]]_{M,tid}\} \text{assume} (\neg x) \{[[\Gamma]]_{M,tid}\}
\]

as the antecedent.

Case 74. -M: where C is a loop statement. C has the form \textit{while}(x.f \neq \text{null})\{C\}

Proof is similar to the one for T-LOOP1.

Case 75. -M: case where C is branch statement. C has the form \textit{if}(e) then\{C_1\} else\{C_2\}.

\[
\Gamma, x : \text{rcultr} \rho N([f_1 \rightarrow z]) \vdash_M C_1 \vdash \Gamma' 
\]
\[
\Gamma, x : \text{rcultr} \rho N([f_2 \rightarrow z]) \vdash_M C_2 \vdash \Gamma'
\]

(392) and (393)

\[
\{[[\Gamma], x : \text{rcultr} \rho N([f_1 \rightarrow z])]]_{M,tid}\} C_1 \{[[\Gamma]]_{M,tid}\}
\]
\[
\{[[\Gamma], x : \text{rcultr} \rho N([f_2 \rightarrow z])]]_{M,tid}\} C_2 \{[[\Gamma]]_{M,tid}\}
\]

(394) and (395)

Our goal is to prove

\[
\{[[\Gamma], x : \text{rcultr} \rho N([f_1 | f_2 \rightarrow z])]\}_{M,tid}\}
\]

\[
y = x.f_1; (\text{assume} (z = y); C_1) + (\text{assume} (y \neq z); C_2)
\]

\[
\{[[\Gamma']]_{M,tid}\}
\]

(396)

where the desugared form includes a fresh variable y. We use fresh variables just for desugaring and they are not included in any type context. We prove (396) from the consequence rule of Views Logic based on the proofs of the following (397) and (398)

\[
\{[[\Gamma], x : \text{rcultr} \rho N([f_1 | f_2 \rightarrow z])]\}_{M,tid}\}
\]

\[
(\text{assume} (z = y); C_1) + (\text{assume} (y \neq z); C_2)
\]

\[
\{[[\Gamma']]_{M,tid}\}
\]

(397)
and
\[ \{ [ \Gamma, x : \text{rultr} \rho \mathcal{N}([f_1 | f_2 \rightarrow z])_{M,tid} \} \]
\[ y = x.f_1 \]
\[ \{ [ \Gamma, x : \text{rultr} \rho \mathcal{N}([f_1 | f_2 \rightarrow z])_{M,tid} ] [ x : \text{rultr} \rho \mathcal{N}([f_1 \rightarrow y]) ]_{M,tid} \} \]

Equation (398) is trivial from the fact that y is a fresh variable and it is not included in any type context and just used for desugaring.

We prove (397) from the branch rule of Views Logic based on the proofs of the following (399) and (400)
\[ \{ [ \Gamma, x : \text{rultr} \rho \mathcal{N}([f_1 | f_2 \rightarrow z]) ]_{M,tid} \}
\[ \{ [ x : \text{rultr} \rho \mathcal{N}([f_1 \rightarrow y]) ]_{M,tid} \}
\[ \text{assume} (z = y) ; C_1 \}
\[ \{ [ \Gamma']_{M,tid} \} \]

and
\[ \{ [ \Gamma, x : \text{rultr} \rho \mathcal{N}([f_1 | f_2 \rightarrow z]) ]_{M,tid} \}
\[ \{ [ x : \text{rultr} \rho \mathcal{N}([f_1 \rightarrow y]) ]_{M,tid} \}
\[ \text{assume} (z \neq y) ; C_2 \}
\[ \{ [ \Gamma']_{M,tid} \} \]

We show (399) from Views Logic’s proof rule for the assume construct by using
\[ \{ [ \Gamma, x : \text{rultr} \rho \mathcal{N}([f_1 | f_2 \rightarrow z]) ]_{M,tid} \}
\[ \{ [ x : \text{rultr} \rho \mathcal{N}([f_1 \rightarrow y]) ]_{M,tid} \}
\[ \text{assume} (y = z) \]
\[ \{ [ \Gamma, x : \text{rultr} \rho \mathcal{N}([f_1 \rightarrow z]) ]_{M,tid} \}
\]

as the antecedent. We can use this antecedent together with
\[ \{ [ \Gamma, x : \text{rultr} \rho \mathcal{N}([f_1 \rightarrow z]) ]_{M,tid} \} C_1 \{ [ \Gamma']_{M,tid} \} \]
as antecedents to the View’s Logic’s proof rule for sequencing.

We show (400) from Views Logic’s proof rule for the assume construct by using
\[ \{ [ \Gamma, x : \text{rultr} \rho \mathcal{N}([f_1 | f_2 \rightarrow z]) ]_{M,tid} \}
\[ \{ [ x : \text{rultr} \rho \mathcal{N}([f_1 \rightarrow y]) ]_{M,tid} \}
\[ \text{assume} (x \neq y) \]
\[ \{ [ \Gamma, x : \text{rultr} \rho \mathcal{N}([f_2 \rightarrow z]) ]_{M,tid} \}
\]
as the antecedent. We can use this antecedent together with
\[ \{ [ \Gamma, x : \text{rultr} \rho \mathcal{N}([f_2 \rightarrow z]) ]_{M,tid} \} C_2 \{ [ \Gamma']_{M,tid} \} \]
as antecedents to the Views Logic’s proof rule for sequencing.

Case 76. -M: case where C is branch statement. C has the form if(x.f == null)then{C_1}else{C_2}. Proof is similar to one for T-BRANCH1.
Case 77. -O: parallel where C has the form $\Gamma_1, \Gamma_2 \vdash C_1 \parallel C_2 \vdash \Gamma_1', \Gamma_2'$ We know

\begin{align*}
\Gamma_1 & \vdash C_1 \vdash \Gamma_1' \\
\Gamma_2 & \vdash C_2 \vdash \Gamma_2' \\
\llbracket\Gamma_1\rrbracket & \llbracket\llbracket C_1\rrbracket\rrbracket \\
\llbracket\Gamma_2\rrbracket & \llbracket\llbracket C_2\rrbracket\rrbracket
\end{align*}

(401) (402) (403) (404)

We need to show

\begin{align*}
\llbracket\Gamma_1, \Gamma_2\rrbracket & \llbracket\llbracket C_1\rrbracket\rrbracket \parallel \llbracket\llbracket C_2\rrbracket\rrbracket \llbracket\llbracket \Gamma_1', \Gamma_2'\rrbracket\rrbracket \\
\llbracket\Gamma_1, \Gamma_2\rrbracket & \llbracket\llbracket C_1\rrbracket\rrbracket \parallel \llbracket\llbracket C_2\rrbracket\rrbracket \llbracket\llbracket \Gamma_1', \Gamma_2'\rrbracket\rrbracket
\end{align*}

(405)

By using (403) and (404) as antecedents to Views Logic's parallel rule, we can draw conclusion for

\begin{align*}
\llbracket\Gamma_1, \Gamma_2\rrbracket & \llbracket\llbracket C_1\rrbracket\rrbracket \parallel \llbracket\llbracket C_2\rrbracket\rrbracket \llbracket\llbracket \Gamma_1', \Gamma_2'\rrbracket\rrbracket \\
\llbracket\Gamma_1, \Gamma_2\rrbracket & \llbracket\llbracket C_1\rrbracket\rrbracket \parallel \llbracket\llbracket C_2\rrbracket\rrbracket \llbracket\llbracket \Gamma_1', \Gamma_2'\rrbracket\rrbracket
\end{align*}

(406)

(406)

Showing (405) requires showing

\begin{align*}
\llbracket\Gamma_1, \Gamma_2\rrbracket & \equiv \llbracket\llbracket C_1\rrbracket\rrbracket \parallel \llbracket\llbracket C_2\rrbracket\rrbracket \\
\llbracket\Gamma_1, \Gamma_2\rrbracket & \equiv \llbracket\llbracket C_1\rrbracket\rrbracket \parallel \llbracket\llbracket C_2\rrbracket\rrbracket
\end{align*}

(407)

(408)

By using (407) and (408) (trivial to show as "," and "+" for denotation of type contexts are both semantically equivalent to $\cap$) as antecedents to Views Logic’s consequence rule, we can conclude (405).

Case 78. -M where C has form RCUWrite $x.f$ as $y$ in $C$ which desugars into

\begin{align*}
\text{WriteBegin; } x.f := y; \text{C; WriteEnd}
\end{align*}

We assume from the rule ToRCUWrite

\begin{align*}
\Gamma, y : \text{rculr} \vdash M C \vdash \Gamma' \\
\text{FType(f)} = \text{RCU} \\
\text{NoFresh}(\Gamma') \\
\text{NoFreeable}(\Gamma') \\
\text{NoUnlinked}(\Gamma') \\
\llbracket\Gamma, y : \text{rculr} \rrbracket M.tid \llbracket\llbracket \Gamma'\rrbracket M.tid\rrbracket
\end{align*}

(409) (410) (411) (412) (413) (414)

Our goal is to prove

\begin{align*}
\llbracket\Gamma M.tid\rrbracket \text{WriteBegin; C; WriteEnd} \llbracket\llbracket \Gamma'\rrbracket M.tid\rrbracket
\end{align*}

(415)

Any case of $C$ does not change the state (no heap update) by assumptions (411)(413) therefore (415) follows from assumptions (409)(414) trivially.
Lemma 24 (Context-SubTyping-M).
\[ \Gamma \llhd \Gamma' \Rightarrow [\Gamma]_{M,tid} \sqsubseteq [\Gamma']_{M,tid} \]

Proof. Induction on the subtyping derivation. Then inducting on the first entry in the non-empty context (empty case is trivial) which follows from 20.

Lemma 25 (Context-SubTyping-R).
\[ \Gamma \llhd \Gamma' \Rightarrow [\Gamma]_{R,tid} \sqsubseteq [\Gamma']_{R,tid} \]

Proof. Induction on the subtyping derivation. Then inducting on the first entry in the non-empty context (empty case is trivial) which follows from 27.

Lemma 26 (Singleton-SubTyping-M).
\[ x : T \llhd x : T' \Rightarrow [x : T]_{M,tid} \sqsubseteq [x : T']_{R,tid} \]

Proof. Proof by case analysis on structure of \( T' \) and \( T \). Important case includes the subtyping relation is defined over components of \( rcuItr \) type. \( T' \) including approximation on the path component
\[ \rho.f_1 \llhd \rho.f_1|f_2 \]

together with the approximation on the field map
\[ \mathcal{N}(\{f_1 \rightarrow \bot\}) \llhd \mathcal{N}(\{f_1|f_2 \rightarrow \bot\}) \]

lead to subset inclusion in between a set of states defined by denotation of the \( x : T' \) the set of states defined by denotation of the \( x : T \) (which is also obvious for T-SUB). Reflexive relations and relations capturing base cases in subtyping are trivial to show.

Lemma 27 (Singleton-SubTyping-R).
\[ x : T \llhd x : T' \Rightarrow [x : T]_{M,tid} \sqsubseteq [x : T']_{M,tid} \]

Proof. Proof is similar to 20 with a single trivial reflexive derivation relation (T-TSub2)
\[ rcuItr \llhd rcuItr \]
void delete(int data) {
    WriteBegin;
    // Find data in the tree
    // Root is never empty and its value is unique id
    BinaryTreeNode current, parent = root;
    {parent : rcuItr e {}}
    current = parent.Right;
    {parent : rcuItr e {Right -> current}}
    {current : rcuItr Right {}}
    while (current! = null&&current.data! = data)
    {
        {parent : rcuItr (Left|Right) RIGHT ((Left|Right) -> current)}
        {current : rcuItr (Left|Right) LEFT ((Left|Right) {})}
        if (current.data > data)
        {
            // if data exists it’s in the left subtree
            parent = current;
            {parent : rcuItr (Left|Right) LEFT {}}
            {current : rcuItr (Left|Right) RIGHT {}}
            current = parent.Left;
            {parent : rcuItr (Left|Right) LEFT {Left -> current}}
            {current : rcuItr (Left|Right) RIGHT {}}
        } else if (current.data < data)
        {
            // if data exists it’s in the right subtree
            parent = current;
            {parent : rcuItr (Left|Right) LEFT {}}
            {current : rcuItr (Left|Right) RIGHT {}}
            current = parent.Right;
            {parent : rcuItr (Left|Right) LEFT {Right -> current}}
            {current : rcuItr (Left|Right) LEFT {}}
        }
    }
    {parent : rcuItr (Left|Right) RIGHT ((Left|Right) -> current)}
    {current : rcuItr (Left|Right) LEFT ((Left|Right) {})}
    // At this point, we’ve found the node to remove
    BinaryTreeNode lmParent = current.Right;
    BinaryTreeNode currentL = current.Left;
    {current : rcuItr(Left|Right) LEFT ((Left|Right)Left -> currentL, Right -> lmParent)}
    {currentL : rcuItr (Left|Right) LEFT ((Left|Right)Left {})}
    {lmParent : rcuItr Left|Right (Right {})}
}
We now need to "rethread" the tree

CASE 1: If current has no right child, then current’s left child becomes the node pointed to by the parent

```csharp
if (current.Right == null)
{
    // parent: rcuItTr (Left|Right)k \{Left|Right\} \rightarrow current\}
    current: rcuItTr (Left|Right)k \{Left\} \rightarrow currentL, Right \rightarrow null\}
    currentL: rcuItTr (Left|Right)k \{Left\} \rightarrow null\}

    // parent.Value is greater than current.Value
    // so make current’s left child a left child of parent
    parent.Left = current;
    \{parent: rcuItTr (Left|Right)k \{Left\} \rightarrow current\}
    \{current: rcuItTr (Left|Right)k \{Left\} \rightarrow currentL, Right \rightarrow null\}
    \{currentL: rcuItTr (Left|Right)k \{Left\} \rightarrow currentL\}

    else
    // parent.Value is less than current.Value
    // so make current’s left child a right child of parent
    parent.Right = current;
    \{parent: rcuItTr (Left|Right)k \{Right\} \rightarrow current\}
    \{current: rcuItTr (Left|Right)k \{Right\} \rightarrow currentL, Right \rightarrow null\}
    \{currentL: rcuItTr (Left|Right)k \{Right\} \rightarrow current\}
    \{current: rcuItTr (Left|Right)k \{Right\} \rightarrow null\}

    SyncStart;
    SyncStop;
    \{current: freeable\}
    Free(current);
    \{current: undef\}
}
```
// CASE 2: If current’s right child has no left child, then current’s right child
// replaces current in the tree
else if (current.Left == null)
{
    parent : rcuItr (Left|Right)^k {(Left|Right) → current}
    current : rcuItr (Left|Right)^k (Left{Left null, Right → lmParent})
    currentL : rcuItr (Left|Right)^k (Left|Right).Left {}}
    lmParent : rcuItr (Left|Right)^k (Left|Right).Right {}}
    lmParent : rcuItr (Left|Right)^k (Left|Right).Left {}}
    if (parent.Left == current)
        parent.Left = lmParent;
        parent : rcuItr (Left|Right)^k {Left → lmParent}
        current : unlinked}
        lmParent : rcuItr (Left|Right)^k .Left {}}
    else
        parent : rcuItr (Left|Right)^k {Right → current})
        current : rcuItr (Left|Right)^k .Right {Left → null, Right → lmParent})
        lmParent : rcuItr (Left|Right)^k .Right {}}
        // parent.Value is less than current.Value
        // so make current’s right child a right child of parent
        parent.Right = lmParent;
        lmParent : rcuItr (Left|Right)^k {Right → lmParent}
        current : unlinked}
        lmParent : rcuItr (Left|Right)^k .Right {}}
        SyncStart;
        SyncStop;
        current : freeable}
        Free(current);
        current : undef}
// CASE 3: If current's right child has a left child, replace current with current's
// right child's left-most descendent
else {
  { parent : rcuItr (Left|Right)k (Left|Right) → current }
  { current : rcuItr (Left|Right)k (Left|Right) → lmParent, Left → currentL }
  { lmParent : rcuItr (Left|Right)k (Left|Right).Right { } }
  { currentL : rcuItr (Left|Right)k (Left|Right).Left { } }
  // We first need to find the right node's left-most child
  BinaryTreeNode currentF = new;
  { currentF : rcuFresh }
  currentF.Right = lmParent;
  { currentF : rcuFresh { Right → lmParent } }
  currentF.Left = currentL;
  { currentF : rcuFresh { Right → lmParent, Left → currentL } }
  BinaryTreeNode leftmost = lmParent.Left;
  { lmParent : rcuItr (Left|Right)k (Left|Right).Right { Left → leftmost } }
  { leftmost : rcuItr (Left|Right)k (Left|Right).Right.Left { } }
  if (lmParent.Left == null) {
    { lmParent : rcuItr (Left|Right)k (Left|Right).Right { Left → null } }
    currentF.data = lmParent.data;
    if (parent.Left == current) {
      { parent : rcuItr (Left|Right)k { Left → current } }
      { current : rcuItr (Left|Right)k.Left { Right → lmParent, Left → currentL } }
      { currentF : rcuFresh { Right → lmParent, Left → currentL } }
      // current's right child a left child of parent
      parent.Left = currentF;
      { parent : rcuItr (Left|Right)k { Left → currentF } }
      { current : unlinked }
      { currentF : rcuItr (Left|Right)k.Left { Right → lmParent, Left → currentL } }
      SyncStart;
      SyncStop;
      { current : freeable }
      Free(current);
      { current : undef }
    }
  else {
    { parent : rcuItr (Left|Right)k { Right → current } }
    { current : rcuItr (Left|Right)k.Right { Right → lmParent, Left → currentL } }
    { currentF : rcuFresh { Right → lmParent, Left → currentL } }
    // current's right child a right child of parent
    parent.Right = currentF;
    { parent : rcuItr (Left|Right)k { Right → currentF } }
    { current : unlinked }
    { currentF : rcuItr (Left|Right)k.Right { Right → lmParent, Left → currentL } }
    SyncStart;
    SyncStop;
    { current : freeable }
    Free(current);
    { current : undef }
  }
}
else{
  \{ lmParent : rcuItr (Left|Right)^k.(Left|Right).Right \{Left \to leftmost\} \}
  \{ leftmost : rcuItr (Left|Right)^k.(Left|Right).Right.Left \{\} \}
while (leftmost.Left != null)
{
  \{ lmParent : rcuItr (Left|Right)^k.(Left|Right).Right.Left\{Left \to leftmost\} \}
  \{ leftmost : rcuItr (Left|Right)^k.(Left|Right).Right.Left\{Left \to leftmost\} \}
  lmParent = leftmost;
  \{ lmParent : rcuItr (Left|Right)^k.(Left|Right).Right.Left\{Left \to leftmost\} \}
  \{ leftmost : rcuItr (Left|Right)^k.(Left|Right).Right.Left\{Left \to leftmost\} \}
  leftmost = lmParent.Left;
  \{ lmParent : rcuItr (Left|Right)^k.(Left|Right).Right.Left\{Left \to leftmost\} \}
  \{ leftmost : rcuItr (Left|Right)^k.(Left|Right).Right.Left\{Left \to leftmost\} \}
}
current.F.data = leftmost.data;
if (parent.Left == current) {
  \{ parent : rcuItr (Left|Right)^k \{Left \to current\} \}
  \{ current : rcuItr (Left|Right)^k \{Right \to lmParent, Left \to current\} \}
  \{ currentF : rcuFresh \{Right \to lmParent, Left \to current\} \}
  //current’s right child a left child of parent
  parent.Left = currentF;
  \{ parent : rcuItr (Left|Right)^k \{Left \to current\} \}
  \{ current : unlinked \}
  \{ currentF : rcuItr (Left|Right)^k \{Left \to current\} \}
  SyncStart;
  SyncStop;
  \{ current : freeable \}
  Free(current);
  \{ current : undef \}
}
else {
  \{ parent : rcuItr (Left|Right)^k \{Right \to current\} \}
  \{ current : rcuItr (Left|Right)^k \{Right \to lmParent, Left \to current\} \}
  \{ currentF : rcuFresh \{Right \to lmParent, Left \to current\} \}
  //current’s right child a right child of parent
  parent.Right = currentF;
  \{ parent : rcuItr (Left|Right)^k \{Right \to current\} \}
  \{ current : unlinked \}
  \{ currentF : rcuItr (Left|Right)^k \{Right \to lmParent, Left \to current\} \}
  SyncStart;
  SyncStop;
  \{ current : freeable \}
  Free(current);
  \{ current : undef \}
}
```plaintext
{lmParent : rcuItr (Left|Right). (Left|Right). Right.Left (Left). Left \{Left \rightarrow leftmost\}}
{leftmost : rcuItr (Left|Right). (Left|Right). Right.Left (Left). Left \{Left \rightarrow null\}}

BinaryTreeNode leftmostR = leftmost.Right;
{leftmost : rcuItr (Left|Right). (Left|Right). Right.Left (Left). Left \{Left \rightarrow null, Right \rightarrow leftmostR\}}
{lmParent : rcuItr (Left|Right). (Left|Right). Right.Left (Left). Left \{Left \rightarrow leftmost\}}
{leftmostR : rcuItr (Left|Right). (Left|Right). Right.Left (Left). Left.Right \{\}}

// the parent's left subtree becomes the leftmost's right subtree
lmParent.Left = leftmostR;
{leftmost : unlinked}
{lmParent : rcuItr (Left|Right). (Left|Right). Right.Left (Left). Left \{Left \rightarrow leftmostR\}}
{leftmostR : rcuItr (Left|Right). (Left|Right). Right.Left (Left). Left \{\}}

SyncStart;
SyncStop;
{leftmost : freeable}
Free(leftmost);
{leftmost : undef}
}
}
WriteEnd;
}```
C RCU Bag with Linked-List

BagNode head;
int member (int toRead) {
    ReadBegin;
    int result = 0;
    {parent : undef, head : rcuRoot}
    BagNode parent = head;
    {parent : rcuItr}
    {current : _}
    current = parent.Next;
    {current : rcuItr, parent : rcuItr}
    {current : rcuItr}
    while (current.data != toRead && current.Next != null){
        {parent : rcuItr}
        {current : rcuItr}
        parent = current;
        current = parent.Next;
        {parent : rcuItr}
        {current : rcuItr}
    }
    {parent : rcuItr}
    {current : rcuItr}
    result = current.data;
    ReadEnd;
    return result;
}
void remove (int toDel) {
    WriteBegin;
    BagNode current, parent = head;
    current = parent.Next;
    {current : rcuItr Next {}}
    {parent : rcuItr ∈ {Next → current}}
    while (current.Next != null & current.data != toDel) {
        {parent : rcuItr (Next)^k {Next → current}}
        {current : rcuItr Next.(Next)^k.Next {}}
        parent = current;
        {current : rcuItr Next.(Next)^k.Next {}}
        {parent : rcuItr Next.(Next)^k.Next {}}
        current = parent.Next;
        {current : rcuItr Next.(Next)^k.Next {Next → current}}
        {current : rcuItr Next.(Next)^k.Next.Next {}}
    }
    // We don’t need to be precise on whether next of current is null or not
    {parent : rcuItr Next.(Next)^k.Next {Next → current}}
    {current : rcuItr Next.(Next)^k.Next.Next.Next {Next → null}}
    BagNode currentL = current.Next;
    {parent : rcuItr Next.(Next)^k.Next {Next → itr}}
    {currentL : rcuItr Next.(Next)^k.Next.Next {}}
    {current : rcuItr Next.(Next)^k.Next.Next {Next → currentL}}
    current.Next = currentL;
    {parent : rcuItr Next.(Next)^k.Next {Next → itrN}}
    {currentL : rcuItr Next.(Next)^k.Next.Next {}}
    {current : unlinked}
    SyncStart;
    SyncStop;
    {current : freeable}
    Free(current);
    {current : undef}
    WriteEnd;
}
void add(int toAdd){
    WriteBegin;
    BagNode nw = new;
    nw.data = toAdd;
    {nw : rcuFresh{}}
    BagNode current, parent = head;
    parent.Next = current;
    {current : rcuItr Next{}}
    {parent : rcuItr ϵ {Next → current}}
    while (current.Next! = null) {
        {parent : rcuItr (Next)² {Next → current}}
        {current : rcuItr Next.(Next)².Next{}}
        parent = current;
        current = parent.Next;
        {parent : rcuItr (Next)².Next {Next → current}}
        {current : rcuItr Next.(Next)².Next.Next{}}
    }
    {parent : rcuItr (Next)³.Next {Next → current}}
    {current : rcuItr Next.(Next)³.Next.Next {Next → null}}
    nw.next = null;
    {nw : rcuFresh {Next → null}}
    current.Next = nw
    {parent : rcuItr (Next)³.Next {Next → nw}}
    {current : rcuItr (Next)³.Next.Next {Next → nw}}
    {nw : rcuItr Next.(Next)³.Next.Next.Next {Next → null}}
    WriteEnd;
}
Preserving invariants of a data structure against possible mutations under RCU semantics is challenging. Unlinking a heap node is one way of mutating the heap. To understand the importance of the locality on the possible effects of the mutation, we illustrate a setting for unlinking a heap in Figures 31a and 31b. The square nodes filled with $R$ – a root node – and $H$ – a heap node – are heap nodes. The hollow nodes are stack pointers to the square heap nodes. All resources in red form the memory footprint of unlinking. The hollow red nodes – $pr$, $cr$ and $crl$ – point to the red square heap nodes which are involved in unlinking of the heap node pointed by $cr$. We have $a_1$, $a_2$ and $a_3$ which are aliases with $parent-pr$, $current-cr$ and $currentL-crl$ respectively. We call them the path-aliases as they share the same path from root to the node that they reference. The red filled circle depicts $null$, $l$ field which depicts $Left$ and $r$ depicts $Right$ field.

The type rule for unlinking must assert the "proper linkage" in between the heap nodes involved in the action of unlinking. We see the proper linkage relation between in Figure 31a as red $l$ links between $H_1$, $H_2$ and $H_3$ which are referenced by $pr$, $cr$ and $crl$ respectively. Our type rule for unlinking(T-UnlinkH) asserts that $x$ (parent), $y$ (current) and $z$ (currentL) pointers are linked with field mappings $N([f_1 \rightarrow z])$ ($Left \mapsto current$) of $x$, $N([f_2 \rightarrow r])$ ($Left \mapsto currentL$) of $y$. In accordance with the field mappings, the type rule also asserts that $x$ has the path $\rho ((Left)^k)$, $y$ has the path $\rho.f_1 ((Left)^k.Left)$ and $z$ has the path $\rho.f_1.f_2 ((Left)^k.Left.Left)$.

Being able to localize the effects of the mutation is important in a sense that it prevents unexpected side effects of the mutation. So, sharing through aliases to the resources under mutation, e.g. aliasing to $parent$, $current$ and $currentL$, needs to be handled carefully. Aliasing can occur via either through object fields
– via field mappings – or stack pointers – via path components. We see path aliases, $a_1$, $a_2$ and $a_3$, illustrated with dashed nodes and arrows to the heap nodes in Figures 31a and 31b. They are depicted as dashed because they are not safe resources to use when unlinking so they are framed-out by the type system via

\[ \neg \text{MayAlias}(\rho_3, \{\rho, \rho_1, \rho_2\}) \]

which ensures the non-existence of the path-aliases to any of $x$, $z$ and $r$ in the rule which corresponds to $pr$, $cr$ and $crl$ respectively.

Any heap node reached from root by following a path($\rho_3$) deeper than the path reaching to the last heap node($crl$) in the footprint cannot be pointed by any of the heap nodes($pr$, $cr$ and $crl$) in the footprint. We require this restriction to prevent inconsistency on path components of references, $\rho_3$, referring to heap nodes deeper than memory footprint $\forall \rho_4 \neq \epsilon. \neg \text{MayAlias}(\rho_3, \rho_2, \rho_4)$.

The reason for framing-out these dashed path aliases is obvious when we look at the changes from the Figure 31a to Figure 31b. For example, $a_1$ points to $H_1$ which has object field Left-$l$ pointing to $H_2$ which is also pointed by current as depicted in the Figure 31a. When we look at Figure 31b, we see that $l$ of $H_1$ is pointing to $H_3$ but $a_1$ still points to $H_1$. This change invalidates the field mapping $\text{Left} \mapsto \text{current}$ of $a_1$ in the rcultr type.

One another safety achieved with framing shows up in a setting where current and $a_2$ are aliases. In the Figure 31b both current and $a_2$ are in the rcultr type and point to $H_2$. After the unlinking occurs, the type of current becomes unlinked although $a_2$ is still in the rcultr type. Framing out $a_2$ prevents the inconsistency in its type under the unlinking operation.

One interesting and not obvious inconsistency issue shows up due to the aliasing between $a_3$ and currentL-crl. Before the unlinking occurs, both currentL and $a_3$ have the same path components. After the unlinking, the path of currentL-crl gets shortened as the path to heap node it points, $H_3$, changes to $\text{(Left)}^k.\text{Left}$. However, the path component of $a_3$ would not change so the path component of $a_3$ in the rcultr would become inconsistent with the actual path reaching to $H_3$.

In addition to path-aliasing, there can also be aliasing via field-mappings which we call field-aliasing. We see field aliasing examples in Figures 31a and 31b $pr$ and $a_1$ are field aliases with $Left - l$ from $H_0$ points to $H_1$, $cr$ and $a_2$ are field aliases with $Left - l$ from $H_4$ points to $H_2$ and $crl$ and $a_3$ are field aliases with $Left - l$ from $H_5$ points to $H_3$. We do not discuss the problems that can occur due to the field-aliasing as they are same with the ones due to path-aliasing. What we focus on is how the type rule prevents field-aliases. The type rule asserts $\wedge (m \notin \{z, r\})$ to make sure that there exists no object field from any other context pointing either to the variable points the heap node that is mutation(unlinking) – current-$cr$ – or to the variable which points to the new $Left$ of parent after unlinking – currentL-crl. We should also note that it is
expected to have object fields in other contexts to point to \( pr \) as they are not in the effect zone of unlinking. For example, we see the object field \( l \) points from \( H_0 \) to \( H_1 \) in Figures 31a and 31b.

Once we unlink the heap node, it cannot be accessed by the new coming reader threads the ones that are currently reading this node cannot access to the rest of the heap. We illustrate this with dashed red \( cr \), \( H_2 \) and object fields in Figure 31b.

Being aware of how much of the heap is under mutation is important, e.g. a whole subtree or a single node. Our type system ensures that there can be only just one heap node unlinked at a time by atomic field update action. To be able to ensure this, in addition to the proper linkage enforcement, the rule also asserts that all other object fields which are not under mutation must either not exists or point to \texttt{null} via

\[
\forall f \in \text{dom}(N_1), f \neq f_2 \implies (N_1(f) = \text{null})
\]
E  Types Rules for RCU Read Section

\[
\frac{\Gamma \vdash_R \alpha \rightarrow \Gamma'}{\Gamma, z : \gamma, x : rcuIttr \vdash z = x : rcuIttr, z : rcuIttr, \Gamma}
\]

\[(T-Reads)\]

\[
\frac{y \notin \text{FV}(\Gamma)}{\Gamma, r : rcuRoot, y : undef \vdash r = y : rcuIttr, r : rcuRoot, \Gamma}
\]

\[(T-Root)\]

\[
\frac{z \notin \text{FV}(\Gamma)}{\Gamma, z : \gamma, x : rcuIttr, N \vdash z = x.f = x : rcuIttr, z : rcuIttr, \Gamma}
\]

\[(T-Read)\]

\[
\frac{\Gamma \vdash_R C \rightarrow \Gamma''}{\Gamma \vdash \text{RCURead} \ x.f \ as \ y \ in \ \{\bar{s}\}}
\]

\[(ToRCURead)\]

\[
\frac{\Gamma \vdash \text{M,R } C \rightarrow \Gamma'}{\Gamma \vdash \text{if}(x) \ then \ C_1 \ else \ C_2 \ \rightarrow \ \Gamma'}
\]

\[(T-Branch)\]

\[
\frac{\Gamma \vdash \text{M,R } C \rightarrow \Gamma'}{\Gamma \vdash \text{T-Skip} \ \rightarrow \ \Gamma'}
\]

\[(T-Skip)\]

\[
\frac{\Gamma \vdash \text{Gamma critical section for RCU Programming}}{\Gamma \vdash \text{skip} \ \rightarrow \ \Gamma'}
\]

\[(T-Skip)\]

\[
\frac{\Gamma \vdash \text{Gamma critical section for RCU Programming}}{\Gamma \vdash \text{skip} \ \rightarrow \ \Gamma'}
\]

\[(T-Skip)\]

\[
\frac{\Gamma \vdash \text{Gamma critical section for RCU Programming}}{\Gamma \vdash \text{skip} \ \rightarrow \ \Gamma'}
\]

\[(T-Skip)\]