Safe Relax 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 one of the most difficult problems 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. Our type system enforces *locality* on heap mutations – one heap node at a time – to preserve *well-formedness* properties of data structures including the well-known *acyclicity*. As part of our soundness proof, we give an abstract semantics for RCU memory management primitives which captures *fundamental law* of RCU [27,2]. To our best knowledge we not only verify a singly linked list and but also a binary search tree for the first time.

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 memory* for nodes to deallocate. At the time a node is unlinked from a data structure, an unknown number of additional threads may have already be using the node, having read a pointer to it before it was unlinked in the heap.

A key insight for manageable solutions to this challenge is to recognize that just as in traditional garbage collection, the unlinked nodes need not be reclaimed immediately, but can instead be reclaimed later after some protocol finishes running. Hazard pointers [30] is 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; readers perform minimal bookkeeping — often nothing they wouldn't already do — and the writer(s) perform additional work to track which readers may have observed a node they wish to deallocate. There are now RCU implementations of many common tree data structures [10,35,6,25,3,21], 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. Sophisticated verification logics can prove correctness of the RCU primitives and clients [18,14,34,23]. But these techniques require significant verification expertise to apply, and are specialized to individual data structures or implementations. 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 new client code.

We propose a type system to ensure that RCU client code uses the RCU primitives correctly. We do this in a general way, not assuming the client implements any specific data structure, only one satisfying some basic properties (like acyclicity) 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 signifiantly simpler than full-blown verification logics
- Demonstration of the type system on two examples: a linked list and a binary search tree
- A proof that the type system guarantees the intended safe use of 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 set [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.[1] It differs, however, in that readers may execute concurrently with the (single) writer.

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

```
1  struct BagNode{
2    int data;
3    BagNode<rcu> Next;
4  }
5  BagNode<rcuRoot> head;
6  void add(int toAdd){
7  WriteBegin;
8  BagNode nw = new;
9  {nw : rcuFresh {}}
10 nw.data = toAdd;
11 {head : rcuRoot, parent : undef, current : undef}
12 BagNode<rcuItr> parent,current = head;
13 {head : rcuRoot, parent : rcultr ϵ {}, current : rcultr ϵ {}}
14 {current : rcultr ϵ {}}
15 current = parent.Next;
16 {current : rcultr Next {}}
17 {parent : rcultr ϵ {Next ↦ current}}
18 while(current.Next != null){
19   {current : rcultr(Next)^k.Next{}}
20   {parent : rcultr(Next)^k{Next ↦ current}}
21   parent = current;
22   current = parent.Next;
23   {current : rcultr(Next)^k.Next.Next{}}
24   {parent : rcultr(Next)^k.Next{Next ↦ current}}
25 }
26 {nw : rcuFresh {}}
27 {current : rcultr (Next)^k.Next {Next ↦ null}}
28 {parent : rcultr (Next)^k {Next ↦ current}}
29 nw.Next= null;
30 {nw : rcuFresh {Next ↦ null}}
31 {current : rcultr (Next)^k.Next {Next ↦ null}}
32 current.Next=nw;
33 {nw : rcultr (Next)^k.Next.Next {Next ↦ null}}
34 {current : rcultr (Next)^k.Next {Next ↦ nw}}
35 WriteEnd;
36 }
```

```
1  void remove(int toDel){
2  WriteBegin;
3  {head : rcuRoot, parent : undef, current : undef}
4  BagNode<rcuItr> parent,current = head;
5  {head : rcuRoot, parent : rcultr ϵ {}, current : rcultr ϵ {}}
6  current = parent.Next;
7  {current : rcultr Next {}}
8  {parent : rcultr ϵ {Next ↦ current}}
9  while(current.Next != null&&current.data != toDel)
10 {
11   {current : rcultr (Next)^k.Next {}}
12   {parent : rcultr (Next)^k {Next ↦ current}}
13   parent = current;
14   current = parent.Next;
15   {current : rcultr (Next)^k.Next.Next {}}
16   {parent : rcultr(Next)^k.Next{Next ↦ current}}
17 }
18 {nw : rcuFresh {}}
19 {parent : rcultr (Next)^k {Next ↦ current}}
20 {current : rcultr (Next)^k.Next {}}
21 BagNode<rcuItr> currentL = current.Next;
22 {current : rcultr (Next)^k.Next {Next ↦ currentL}}
23 {currentL : rcultr (Next)^k.Next.Next {}}
24 prev.Next = currentL;
25 {parent : rcultr (Next)^k {Next ↦ currentL}}
26 {current : unlinked}
27 {currentL : rcultr (Next)^k.Next {}}
28 SyncStart;
29 SyncStop;
30 {current : freeable}
31 Free(current);
32 {current : undef}
33 WriteEnd;
34 }
```

Fig. 1: Read-Copy-Update Client: Linked-List based Bag Implementation.

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/deleting nodes to a set(whole type cheking including membership queries for bag can be found in C). Algorithmically, this code is nearly the same as any sequential implementation. There are only two differences. First, the read-side critical section in `member` is indicated by the use of `ReadBegin` and `ReadEnd`; the write-side critical section is between `WriteBegin` and `WriteEnd`. Second, rather than immediately reclaiming the memory for the unlinked node, `remove` calls `SyncStart` to begin a *grace period* — a wait for reader threads that may still hold references to unlinked nodes to finish their critical sections. `SyncStop` blocks execution of the reader 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 for that thread.

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 modifying the structure *or the writer structures used for grace periods.*

`SyncStart` and `SyncStop` implement *grace periods* [32]: a mechanism to wait for readers to finish with any nodes the writer may have unlinked. A grace period begins when a writer requests one, and finishes when all reader threads active *at the start of the grace period* have finished their current critical section. Any nodes a writer unlinks before a grace period are physically unlinked, but not logically unlinked until after one grace period. ⟦Iso says: An astute reader might already realize that our usage of logical/physical unlinking is different than the one used in data-structures literatur 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. Note that `SyncStop` 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 `SyncStart-SyncStop`'s critical section, it is possible that A may exit and re-enter a read-side critical section before B exits, and vice versa. Implementations must distinguish subsequent read-side critical sections from earlier ones that overlapped the writer's initial request to wait: since `SyncStart` is used *after* a node is physically removed from the data structure and readers may not retain RCU references across critical sections, A re-entering a fresh read-side critical section will not permit it to re-observe the node to be freed.

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* semantics for RCU implementations — horrifically 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 critial 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:

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

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 contains:

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

The lock l enforces mutual exclusion between write-side critical sections. The root location rt is the root of an RCU data structure. We model only a single global RCU data structure, 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 `WriteBegin` and `WriteEnd`, which acquire and release the lock that enforces mutual exclusion between writers. `WriteBegin` only reduces (acquires) if the lock is unlocked.

Standard RCU APIs include a primitive `synchronize_rcu()` to wait for a grace period for the current readers. We decompose this here into two actions,

$$a ::= \mathsf{skip} \mid \mathsf{x.f} = \mathsf{y} \mid \mathsf{y} = \mathsf{x} \mid \mathsf{y} = \mathsf{x.f} \mid \mathsf{y} = \mathsf{new} \mid \mathsf{Free(x)} \mid \mathsf{Sync}$$
$$\mathsf{RCURead\ x.f\ as\ y} \in \{\bar{s}\} \mid \mathsf{RCUWrite\ x.f\ as\ y} \in \{\bar{s}\}$$

$$[\![\mathsf{RCUWrite\ x.f\ as\ y\ in}\ \{\bar{s}\}]\!]_{\Downarrow \mathrm{tid}} \stackrel{\Delta}{=} \mathsf{WriteBegin\ ;\ x.f\ :=\ y\ ;WriteEnd}$$
$$[\![\mathsf{RCURead\ x.f\ as\ y\ in}\ \{\bar{s}\}]\!]_{\Downarrow \mathrm{tid}} \stackrel{\Delta}{=} \mathsf{ReadBegin;\ y:=\ x.f\ ;ReadEnd}$$
$$[\![\mathsf{Sync}]\!]_{\Downarrow \mathrm{tid}} \stackrel{\Delta}{=} \mathsf{SyncStart;SyncStop}$$

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

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

$$\begin{array}{rrll}
(\text{RCU-W-Begin}) & [\![\texttt{WriteBegin}]\!] & (s,h,\mathsf{unlocked},rt,R,B) & \Downarrow_{tid}(s,h,l,rt,R,B) \\
(\text{RCU-W-End}) & [\![\texttt{WriteEnd}]\!] & (s,h,l,rt,R,B) & \Downarrow_{tid}(s,h,\mathsf{unlocked},rt,R,B) \\
(\text{RCU-R-Begin}) & [\![\texttt{ReadBegin}]\!] & (s,h,tid,rt,R,B) & \Downarrow_{tid}(s,h,tid,rt,R \uplus \{tid\},B)\ \ tid \neq l \\
(\text{RCU-R-End}) & [\![\texttt{ReadEnd}]\!] & (s,h,tid,rt,R \uplus \{tid\},B) & \Downarrow_{tid}(s,h,l,rt,R,B \setminus \{tid\})\ \ tid \neq l \\
(\text{RCU-Synch-Start}) & [\![\texttt{SyncStart}]\!] & (s,h,l,rt,R,\emptyset) & \Downarrow_{tid}(s,h,l,rt,R,R) \\
(\text{RCU-Synch-Stop}) & [\![\texttt{SyncStop}]\!] & (s,h,l,rt,R,\emptyset) & \Downarrow_{tid}(s,h,l,rt,R,\emptyset) \\
(\textsc{Free}) & [\![\texttt{Free}(x)]\!] & (s,h,l,rt,R,B) & \Downarrow_{tid}(s,h,l,rt,R,B)
\end{array}$$

Fig. 2: Operational Semantics for RCU

`SyncStart` and `SyncStop`. `SyncStart` initializes the blocking set to the current set of readers — the threads that may have already observed any nodes the writer has unlinked. `SyncStop` blocks until the blocking set is emptied by completing reader threads.

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

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

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

4 Type System & Programming Language

In this section, we present a simple object oriented programming language with two block constructs for modelling RCU, and a type system that ensures proper (memory-safe) use of the language. To ensure memory safety, we must show how our type system makes programs preserve the following properties:

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

We also demonstrate that the type system is not only sound, but useful: we show how it types Figure 1's list-based bag implementation [26]. We also give type checked fragments of a binary search tree to motivate advanced features of the type system; the full typing derivation can be found in Appendix B. The BST

requires type narrowing operations that refine a type based on dynamic checks (e.g., determining which of several fields links to a node). In our system, we presume all objects contain all fields, but the number of fields is finite (and in our examples, small). This avoids additional overhead from tracking well-established aspects of the type system — class and field types and presence, for example — and focus on checking correct use of RCU primitives. Essentially, we assume the code our type system applies to is already type-correct for a system like C or Java's type system.

4.1 RCU Type System for Write Critical Section

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

RCU Types There are six types used in Write critical sections

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

rcuItr is the type given to references pointing into a shared RCU data structure. A rcultr 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 rcultr reference is reachable by following the path ρ from the root. A component $\mathcal{N}$ is a set of field mappings taking the field name to local variable names. Field maps are extended when the referent's fields are read. The field map and path components track reachability from the root, and local reachability between nodes. These are used to ensure the structure remains acyclic, and for the type system to recognize exactly when unlinking can occur.

Read-side critical sections use rcultr without path or field map components. These components are both unnecessary for readers (who perform no updates) and would be invalidated by writer threads anyways. The read-side rules essentially only ensure the reader performs no writes, so we show all type rules related to Read critical section in 31 Appendix E together with the other standard rules (T-Exchange, T-Par, T-Seq, T-Conseq and T-Skip).

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 aquired 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 pointing at unlinked heap location which 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.

$$\mathcal{N}_\emptyset = \{\} \quad \mathcal{N}_{f,\emptyset} = \mathcal{N} \setminus \{f \rightharpoonup \_\} \quad \mathcal{N}(\cup_{f \rightharpoonup y}) = \mathcal{N} \cup \{f \rightharpoonup y\} \quad \mathcal{N} = \{f \rightharpoonup y \mid f \in \mathsf{FName} \wedge (y \in \mathsf{Var} \vee y \in \{null\})\}$$

$$\mathcal{N}(\setminus_{f \rightharpoonup y}) = \mathcal{N} - \{f \rightharpoonup y\} \qquad \mathcal{N}([f \rightharpoonup y]) = \mathcal{N} \text{ where } f \rightharpoonup y \in \mathcal{N}$$

$$\mathcal{N}(f \rightharpoonup x \setminus y) = \mathcal{N} \setminus \{f \rightharpoonup x\} \cup \{f \rightharpoonup y\}$$

$\boxed{\vdash \mathcal{N} \prec: \mathcal{N}'}$

$$\text{T-NSub3}\ \frac{}{\vdash \mathcal{N}_{f,\emptyset} \prec: \mathcal{N}([f \rightharpoonup y])} \qquad \text{T-NSub4}\ \frac{}{\vdash \mathcal{N}_\emptyset \prec: \mathcal{N}} \qquad \text{T-NSub5}\ \frac{}{\vdash \mathcal{N} \prec: \mathcal{N}}$$

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

$\boxed{\vdash \rho \prec: \rho'}$

$$\text{T-PSub1}\ \frac{}{\vdash \rho.f_1 \prec: \rho.f_1|f_2} \qquad \text{T-PSub2}\ \frac{}{\vdash \rho.f_2 \prec: \rho.f_1|f_2} \qquad \text{T-PSub3}\ \frac{}{\vdash \rho \prec: \rho}$$

$\boxed{\vdash T \prec: T'}$

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

$\boxed{\vdash \Gamma \prec: \Gamma'}$

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

Fig. 3: Sub-Typing Judgements

undef is the type given to references where the content of the referenced location is inaccessible. A freeable object becomes undefined after it is reclaimed.

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

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

Subtyping It is sometimes necessary to use imprecise types — mostly for control flow joins. Our type system performs these abstractions via subtyping on individual types and full contexts, as in Figure 3. We discuss subtyping before the full type rules to give a sense of what kinds of imprecision exist in our type system.

Figure 3 includes four judgments. The first two — $\vdash \mathcal{N} \prec: \mathcal{N}'$ and $\vdash \rho \prec: \rho'$ — describe relaxations of field maps and paths respectively. $\vdash \mathcal{N} \prec: \mathcal{N}'$ is read as "the field map $\mathcal{N}$ is more precise than $\mathcal{N}'$" and similarly for paths. The third judgment $\vdash T \prec: T'$ uses path and field map subtyping to give subtyping among rcultr types — one rcultr is a subtype of another if their paths and field maps are similarly more precise — and to allow rcultr references to be "forgotten" — this is occasionally needed to satisfy non-interference checks in the type rules. The final judgment $\vdash \Gamma \prec: \Gamma'$ extends subtyping to all assumptions in a type context, allowing any or all types in the context to be relaxed to a supertype.

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

$$\boxed{\Gamma \vdash_M \bar{s} \dashv \Gamma'} \quad \text{(T-Branch1)} \ \frac{\Gamma, x : \mathsf{rcultr}\,\rho\,\mathcal{N}([f_1 \rightharpoonup z]) \vdash \bar{s}_1 \dashv \Gamma_4 \qquad \Gamma, x : \mathsf{rcultr}\,\rho\,\mathcal{N}([f_2 \rightharpoonup z]) \vdash \bar{s}_2 \dashv \Gamma_4}{\Gamma, x : \mathsf{rcultr}\,\rho\,\mathcal{N}([f_1 \mid f_2 \rightharpoonup z]) \vdash \mathsf{if}(x.f_1 == z)\ \mathsf{then}\ \bar{s}_1\ \mathsf{else}\ \bar{s}_2 \dashv \Gamma_4}$$

$$\text{(T-Branch3)} \ \frac{\Gamma, x : \mathsf{rcultr}\,\rho\,\mathcal{N}([f \rightharpoonup y \setminus \mathsf{null}]) \vdash \bar{s}_1 \dashv \Gamma' \qquad \Gamma, x : \mathsf{rcultr}\,\rho\,\mathcal{N}([f \rightharpoonup y]) \vdash \bar{s}_2 \dashv \Gamma'}{\Gamma, x : \mathsf{rcultr}\,\rho\,\mathcal{N}([f \rightharpoonup y]) \vdash \mathsf{if}(x.f == \mathsf{null})\ \mathsf{then}\ \bar{s}_1\ \mathsf{else}\ \bar{s}_2 \dashv \Gamma'}$$

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

$$\text{(T-ReIndex)} \ \frac{}{\Gamma \vdash \bar{s}_k \dashv \Gamma[\rho.f^k / \rho.f^k.f]}$$

Fig. 4: Type Judgements for Control-Flow.

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 Listing 1.1, we see that `current` aliases different fields of `parent` — either $Left$ or $Right$ — in different branches of the conditional. The types after the conditional must overapproximate this, here as $Left|Right \mapsto current$ in `parent`'s field map, and a similar path disjunction in `current`'s path. This is reflected in Figure 3's T-NSub1-5 and T-PSub1-2 — within each branch, each type is coerced to a supertype to validate the control flow join.

Listing 1.1: Choosing fields to read

```
1  {current : rcultr Left|Right {},  parent : rcultr ϵ {Left|Right ↦ current}}
2  if(parent.Left == current){
3    {current : rcultr Left {},  parent : rcultr ϵ {Left ↦ current}}
4    parent = current;
5    current = parent.Left;
6    {current : rcultr Left.Left {},  parent : rcultr Left {Left ↦ current}}
7  }else{
8    {current : rcultr Right {},  parent : rcultr ϵ {Right ↦ current}}
9    parent = current;
10   current = parent.Right;
11   {current : rcultr Right.Right {},  parent : rcultr Right {Right ↦ current}}
12 }{current : rcultr Left|Right.Left|Right {},  parent : rcultr Left|Right {Left|Right ↦ current}}
```

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

Assertions for `add` function in lines 19 and 20 in Figure 1 show the *loop*'s effects on paths of iterator references used inside the loop, `current` and `parent`. On line 20, `parent`'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 `current`'s path on line 19 — $(Next)^k.Next$ — expresses the relationship between `current` and `parent`: `parent` is reachable from the root by following $Next$ k times, and `current` is reachable via one additional $Next$. The types of 19 and 20, however, are not the same as lines 23 and 24, so an additional adjustment is needed for the types to become loop-invariant. *Reindexing* (T-ReIndex in Figure 4) effectively increments an abstract loop counter, contracting $(Next)^k.Next$ to $Next^k$ everywhere in a type environment. This expresses the same relationship between `parent` and `current` 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.

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 Listing 1.1, we see a conditional with the condition $parent.Left == current$. 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 3 and 8) based on whether execution reflects the truth or falsity of that check. Similarly, it is sometimes required to check — and later remember — whether a field is null, and the type system supports this.

4.2 Types in Action

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

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

Reading a Global RCU Root All RCU data structures have fixed roots, which we characterize with the rcuRoot type. Each operation in Figure 1 begins by reading the root into a new rcuItr reference used to begin traversing the

structure. After each initial read (line 12 of `add` and line 4 of `remove`), the path of `current` reference is the empty path (ϵ) and the field map is empty ($\{\}$), because it is an alias to the root, and none of its field contents are known yet.

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

Unlinking Nodes Line 24 of `remove` in Figure 1 unlinks a node. The type annotations show that before that line `current` 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 `parent`, `current`, and `currentL` just before line 24 completely describe a section of the list.

Grace and Reclamation After the referent of `current` 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 `current`'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 `current` to undef reflecting that `current` is no longer a valid reference. The type system also ensures no local (writer) aliases exist to the freed node.

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 `current` remain reachable: note the field maps of `current` and `nw` in lines 30–31 are equal (trivially, though in general the field need not be null).

4.3 Type Rules

Figure 5 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

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

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

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

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

(T-WriteFH)

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

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

(T-UnlinkH)

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

(T-LinkF)

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

$$\text{(ToRCUWrite)}\ \frac{\Gamma, y : \mathsf{rcultr}\ \_ \vdash_M \bar{s} \dashv \Gamma' \quad \mathsf{FType}(f) = \mathsf{RCU} \quad \mathsf{NoFresh}(\Gamma') \quad \mathsf{NoUnlinked}(\Gamma')}{\Gamma \vdash \mathsf{RCUWrite}\, x.f \text{ as } y \text{ in } \{\bar{s}\}}$$

Fig. 5: Type Rules for Write side critical section

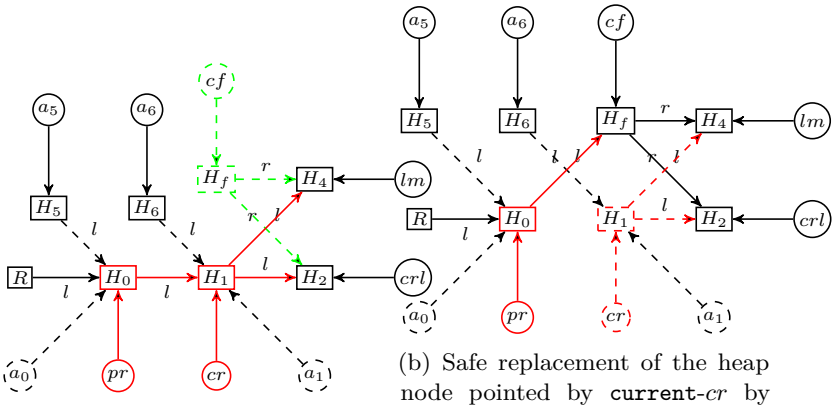
reflect that the overwritten variable z holds the same value as $x.f$. T-WriteFH updates a field of a *fresh* (thread-local) object, similarly tracking the update in the fresh object's field map at the type level. The remaining rules are a bit more involved, and form the heart of the type system.

Grace Periods T-Sync gives pre- and post-environments to the compound statement `SyncStart;SyncStop`, which implements a grace period. As mentioned earlier, this works in any type environment, and updates the environment afterwards to reflect that any nodes unlinked before the wait become freeable after the wait.

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.

Finally, the rule must ensure that no types in Γ are invalidated. This could happen one of two ways: either a field map in Γ for an alias of x duplicates the assumption that $x.f_1 == z$ (which is changed by this write), or Γ contains a descendant of r, whose path from the root will change when its ancestor is modified. The final assumption of T-UnlinkH (the implication) checks that for every rcultr reference n in Γ, it is not a path alias of x, z, or r; no entry of its field map (m) refers to r or z (which would imply n aliased x or z initially); and its path is not an extension of r (i.e., it is not a descendant). MayAlias is a predicate on two paths (or a path and set of paths) which is true if it is possible that any concrete paths the arguments may abstract (e.g., via adding non-determinism through | or abstracting iteration with indexing) *could* be the same. The negation of a MayAlias use is true only when the paths are guaranteed to refer to different locations in the heap.



(a) Safe linking of the *fresh* heap node pointed by `currentF-`cf

(b) Safe replacement of the heap node pointed by `current-`cr by the *fresh* heap node pointed by `currentF-`cf

Fig. 6: Linking and replacing nodes

Linking a Fresh Node Linking a rcuFresh reference faces the same aliasing complications as direct unlinking. We illustrate these challenges in Figures 6a and 6b. The square R nodes are root nodes, and H nodes are general heap nodes. All resources in red and green form the memory foot print of unlinking. The hollow red circular nodes – pr and cr – point to the nodes involved in replacing H_1 (referenced by `cr`) wih H_f (referenced by cf) in the structure. We 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 6a) 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 rcuFresh. Note that we cannot duplicate a rcuFresh variable, nor read a field of an object it points to. This restriction localizes our reasoning about the effects of linking 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 6a to 6b illustrates the effects of the heap mutation (linking a fresh node). The reasoning in the type system for linking a fresh node is nearly the same as for unlinking an existing node, with one exception. In linking 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 linking a fresh node, those descendants' paths are preserved. Our type rule for ensuring safe linking (T-LinkF) prevents path aliasing (nonexistence of a_0 and a_1) by negating a MayAlias query and prevents field mapping aliasing (nonexistence of any object field from any other context pointing to cr) via asserting $(y \neq o)$. It is important to note that objects(H_4, H_2) in the field mappings of the cr whose referent is to be unlinked captured by the heap node's field mappings referenced by cf in rcuFresh. This is part of enforcing locality on the heap mutation and captured by assertion $\mathcal{N} = \mathcal{N}'$ in the type rule(T-LinkF).

There is also another rule, T-LinkF-Null, not shown in Figure 5, 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(ToRCURead in Figure 31 Appendix E and ToRCUWrite in Figure 5) 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 multiple

writers, while ours uses only one). 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.

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

6.1 Proof Outline

Here we outline our proof type soundness. Our full proof appears in the appendices.

We prove type soundness by embedding the type system into an abstract concurrent separation logic called the Views Framework [11]. As with other work taking this approach [17,16], this consists of several key steps:

1. Provide runtime states and semantics for the atomic actions we care about in the language. These are exactly the semantics from Section 3.
2. Define a denotational interpretation $[\![-]\!]$ of the types in terms of an 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, or that freeable references are unreachable from the heap as well as other threads' local variables. The instrumented execution states are also given additional global invariants the type system is intended to maintain, such as acyclicity of the structure.
3. Prove a lemma — called *Axiom Soundness* — that the type rules in Section 4.3 are consistent. 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 of non-trivial control constructs into the simpler nondeterministic versions provided by Views.

The top-level soundness claim takes the form $\forall \Gamma, C, \Gamma', \Gamma \vdash_M C \dashv \Gamma' \Rightarrow \{[\![\Gamma]\!]\} \downarrow C \downarrow \{[\![\Gamma']\!]\}$. The judgment $\{-\}\ C\ \{-\}$ is a claim in the abstract separation logic provided by the Views Framework, which is sound as long as the earlier steps satisfy some natural consistency properties (this is a result of the Views Framework itself [11]). 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.

6.2 Informal Invariants of RCU Model and Denotations of Types

We define *well-formed*ness condition of RCU model through a conjuction of *global invariants* that we define formally in Appendix A.2. Here we introduce the important aspects RCU invariants together with the type denotations – formal denotations can be found in Figure 7 Section 6.3 – to give the intuition that our system captures the requirements of RCU setting [27].

Grace-Period Guarantee A mutator thread has to synchronize with all of the reader threads currently running the read-side critical section to *reclaim* the heap locations. The operational semantics in Figure 2 Section 3 enforce a *protocol* on the mutator thread's actions :1-first it updates a heap location(e.g. *unlinking*) 2-then it *synchronizes* with reader threads and then 3-*reclaims* the heap location. The mutator thread's reference to the heap location in between *unlinking* and *synchronising* is represented with the type unlinked. The type of the reference in between the end of *synchronisation* and *reclamation* is represented with the type freeable. Synchronisation action *bounds* the mutator thread from reclaiming the *unlinked* heap location immediately by adding it to the *free-list*. The denotation of unlinked in Figure 7 Section 6.3 asserts that if an unlinked heap location is in the *free-list* then there exists a subset of reader threads running simultaneously – which are called *bounding* threads – referencing to the heap location. Denotation of freeable asserts that all bounding threads left the RCU read-side critical section – ready to perform reclamation. Iterators-Free-List invariant and Readers-In-Free-List invariant in Appendix A.2 assert what being a *bounding-thread* is, how a *bounding-thread* observes the unlinked heap location and the *bounding-thread*'s existence in *free-list*.

Publish-Subscribe Guarantee Concurrent reader threads must not be affected by a *replacing-an-existing-node-with-a-fresh-one* or *linking-a-single-node* action when inserting a new node into the linked data structure in such a way that they cannot access to the fresh nodes until they are published/linked. Unlike the case in *Grace-Period-Guarantee* where the reader threads can observe an unlinked heap node as iterator during the *grace-period*, they cannot observe the fresh heap nodes until they are published(can be observed as iterator). As we see in the operational semantics in Figure 2 Section 3, once a new heap location is allocated it is in type fresh – observed as fresh by mutator. Fresh-Writes invariant in Appendix A.2 asserts that a fresh heap location can only be allocated and referenced by the mutator thread. Fresh-Reachable invariant asserts the relation between a freshly allocated heap and the rest of the heap in such a way that there exists no heap node reaching to the freshly allocated one. Moreover, Fresh-Not-Reader invariant asserts that freshly allocated heap locatations are not published so they cannot be observed by a reader thread. Our system ensures *locality* of the effects in linking a new heap node via replacing an existing one as discussed in Figure 6b Section 4.3. To do so the fields of the fresh node has to point the nodes which are pointed by the replaced one – Fresh-Points-Iterator invariant.

Memory-Barrier Guarantee RCU write-side critical section has to guarantee that none of the updates to the memory cause no invalid memory access. If a heap location is observed as iterator by the mutator thread then it must

be observed as iterator by other threads – invariant Writer-Unlink and all of the heap locations reaching to the node starting from root can be observed as iterator as well – denotation $[\![\mathsf{rcultr}\,\rho\,\mathcal{N}]\!]_{tid}$ in Figure 7 Section 6.3. Reader threads can only have one type of observation iterator – invariant Readers-Iterator-Only in Appendix A.2 – for a heap location where this heap location can be updated by the mutator and reader threads referencing on this heap node become *bounding-threads* for the reclamation of the heap location – denotation $[\![\mathsf{rcultr}]\!]_{tid}$ and invariant Readers-In-Free-List. However, *bounding-threads* are the only ones which can observe the unlinked heap location as iterator: new coming reader threads cannot access to the unlinked heap. There exists links to mutated heap nodes – e.g. either unlinked or freeable – only from the heap nodes that are already mutated – Unlinked-Reachability and Free-List-Reachability invariants. To sum up, every heap node accessible via traversal starting from the unique root must be valid and observed as iterator both by reader threads and the mutator thread.

RCU Primitives Guaranteed to Execute Unconditionally Unconditional execution of RCU Primitives are provided by definition of operational semantics of RCU primitives.

Guaranteed Read-to-Write Upgrade Our system provides a clear separation of *traverse-and-update* and *traverse-only* intentions for RCU programming through the type system together with the programming primitives – e.g. `ReadBegin`-`ReadEnd` and `WriteBegin`-`WriteEnd`.

Acyclicity of Linked Data Structures Additionally, our system ensures that none of the heap updating actions introduce *acyclicity*. – Unique-Reachable invariant, Unique-Root invariant and denotation $[\![\mathsf{rcultr}\,\rho\,\mathcal{N}]\!]_{tid}$.

6.3 Proof

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 5 Section 4.2 and 31 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 $\mathsf{Loc} \to \mathcal{P}(\mathsf{obs})$

– Undefined variable map, U, of type $\mathcal{P}(\mathsf{Var} \times \mathsf{TID})$
– Set of threads, T, of type $\mathcal{P}(\mathsf{TIDS})$
– A to-free map(or free list), F, of type $\mathsf{Loc} \rightharpoonup \mathcal{P}(\mathsf{TID})$

The 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

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

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

$$\mathsf{RCU} = \{o \mid \mathsf{ftype}(f) = \mathsf{RCU} \wedge \exists o'.\, h(o', f) = o\}$$

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 and F)

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

We do our reasoning for soundness over instrumented states and define an erasure relation

$$\lfloor - \rfloor : \mathsf{MState} \implies \mathsf{LState}$$

that projects instrumented states to the common components with MState.

Well-formedness imposes restrictions over the representation of the RCU structure in memory and type of threads for actions defined in operational semantics. For instance, *an unlinked heap location cannot be reached from a root of an RCU data structure*. We define well-formedness as conjuction of memory axioms defined in Appendix Section A.2.

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

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

that yields the set of states corresponding to a given type environment. This is defined in terms of denotation of individual variable assertions

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

The latter is given in Figure 7. To define the former, we first need to state what it means to combine logical machine states.

Composition of instrumented states is an operation

$$\bullet : \mathcal{M} \longrightarrow \mathcal{M} \longrightarrow \mathcal{M}$$

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

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

Fig. 7: Type Environments

$$
\begin{aligned}
\bullet = (\bullet_\sigma, \bullet_O, \cup, \cup) \quad & O_1 \bullet_O O_2(loc) \stackrel{\text{def}}{=} O_1(loc) \cup O_2(loc) \\
(s_1 \bullet_s s_2) \stackrel{\text{def}}{=} & \; s_1 \cup s_2 \;\; \texttt{when} \;\; dom(s_1) \cap dom(s_2) = \emptyset \\
(F_1 \bullet_F F_2) \stackrel{\text{def}}{=} & \; F_1 \cup F_2 \;\; \texttt{when} \;\; dom(F_1) \cap dom(F_2) = \emptyset \\
(h_1 \bullet_h h_2)(o,f) \stackrel{\text{def}}{=} & \begin{cases} \text{undef} & \text{if } h_1(o,f) = v \wedge h_2(o,f) = v' \wedge v' \neq v \\ v & \text{if } h_1(o,f) = v \wedge h_2(o,f) = v \\ v & \text{if } h_1(o,f) = \text{undef} \wedge h_2(o,f) = v \\ v & \text{if } h_1(o,f) = v \wedge h_2(o,f) = \text{undef} \\ \text{undef} & \text{if } h_1(o,f) = \text{undef} \wedge h_2(o,f) = \text{undef} \end{cases} \\
((s,h,l,rt,R,B),O,U,T,F) \mathcal{R}_0 ((s',h',l',rt',R',B'),O',U',T',F') \stackrel{\text{def}}{=} & \\
\bigwedge \begin{cases} l \in T \rightarrow (h = h' \wedge l = l') \\ l \in T \rightarrow F = F' \\ \forall tid, o.\, \mathsf{iterator}\, tid \in O(o) \rightarrow o \in dom(h) \\ \forall tid, o.\, \mathsf{iterator}\, tid \in O(o) \rightarrow o \in dom(h') \\ \forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h) \\ \forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h') \\ O = O' \wedge U = U' \wedge T = T' \wedge R = R' \wedge rt = rt' \\ \forall x, t \in T.\, s(x,t) = s'(x,t) \end{cases}
\end{aligned}
$$

Fig. 8: Composition($\bullet$) and Thread Interference Relation($\mathcal{R}_0$)

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 8 An important property of composition is that it preserves validity of logical states:

Lemma 1 (Well Formed Composition). *Any successful composition of two well-formed logical states is well-formed:*

$$\forall_{x,y,z}.\, \mathsf{WellFormed}(x) \implies \mathsf{WellFormed}(y) \implies x \bullet y = z \implies \mathsf{WellFormed}(\mathsf{z})$$

Proof. In Appendix A.1.

We define separation on elements of type contexts

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

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

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

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 $\mathcal{R}$ on $\mathcal{M}$, and a View in the formal framework is then:

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

Thread interference relation

$$\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$$

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

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

where $\mathcal{R}$ is transitive-reflexive closure of $\mathcal{R}_0$ shown at Figure 8. $\mathcal{R}_0$ (and therefore $\mathcal{R}$) also "preserves" validity:

Lemma 2 (Valid $\mathcal{R}_0$ Interference). *For any m and m', if* $\mathsf{WellFormed}(m)$ *and $m\mathcal{R}_0 m'$, then* $\mathsf{WellFormed}(m')$.

Proof. In Appendix A.1.

Lemma 3 (Stable Environment Denotation-M). *For any* closed *environment* Γ *(i.e.,* $\forall x \in \mathsf{dom}(\Gamma).\, \mathsf{FV}(\Gamma(x)) \subseteq \mathsf{dom}(\Gamma)$*):*

$$\mathcal{R}(\llbracket \Gamma \rrbracket_{\mathsf{M},tid}) \subseteq \llbracket \Gamma \rrbracket_{\mathsf{M},tid}$$

Alternatively, we say that environment denotation is stable *(closed under* $\mathcal{R}$*).*

Proof. In Appendix A.1.

Lemma 4 (Stable Environment Denotation-R). *For any* closed *environment* Γ *(i.e.,* $\forall x \in \mathsf{dom}(\Gamma).\, \mathsf{FV}(\Gamma(x)) \subseteq \mathsf{dom}(\Gamma)$*):*

$$\mathcal{R}(\llbracket \Gamma \rrbracket_{\mathsf{R},tid}) \subseteq \llbracket \Gamma \rrbracket_{\mathsf{R},tid}$$

Alternatively, we say that environment denotation is stable *(closed under* $\mathcal{R}$*).*

Proof. In Appendix A.1.

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:

$$\forall \Gamma, C, \Gamma', tid.\, \Gamma \vdash_{M,R} C \dashv \Gamma' \Rightarrow \{\llbracket \Gamma \rrbracket_{tid}\} \llbracket C \rrbracket_{tid} \{\llbracket \Gamma' \rrbracket_{tid}\}$$

The antecedent of the implication is a type judgment shown in Figures 5 Section 4.3, 4 Section 4.1 and 31 Appendix E, and the conclusion is a judgment in the Views logic. The environments are translated to views ($\mathsf{View}_{\mathcal{M}}$) 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 9. $\mathsf{assume}(b)$ is a standard construct in verification semantics [4] [31], which "does nothing" (freezes) if the condition b is false, so its postcondition in the Views logic can reflect the truth of b. 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 α ensuring that the operation interpretation of the action satisfies the specification:$p \sqsubseteq q \overset{def}{\Leftrightarrow} \forall m \in \mathcal{M}.\, \lfloor p * \{m\} \rfloor \subseteq \lfloor q * \mathcal{R}(\{m\}) \rfloor$. 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

$$[\![\mathsf{if}\ (x.f == y)\ C_1\ C_2]\!]_{tid} = \\ z = x.f; ((\mathsf{assume}(z = y); C_1) + (\mathsf{assume}(z \neq y); C_2))$$

$$[\![\mathtt{assume}(\mathcal{S})]\!](s) \stackrel{\text{def}}{=} \begin{cases} \{s\} \text{ if } s \in \mathcal{S} \\ \emptyset \quad \text{Otherwise} \end{cases} \quad \frac{\{P\} \cap \{\lceil \mathcal{S} \rceil\} \sqsubseteq \{Q\}}{\{P\}\mathtt{assume}\,(b)\,\{Q\}}\ \mathsf{where}$$

$$\lceil \mathcal{S} \rceil = \{m | \lfloor m \rfloor \cap \mathcal{S} \neq \emptyset\}\ [\![\mathsf{while}\ (e)\ C]\!]_{tid} = (\mathsf{assume}(e); C)^*; (\mathsf{assume}(\neg e));$$

Fig. 9: Encoding of assume(b)

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 $\Gamma <: \Gamma'$, then $[\![\Gamma]\!] \sqsubseteq [\![\Gamma']\!]$. And finally, we must prove that each atomic action's type rule corresponds to a valid semantic judgment in the Views Framework:

$$\forall m.\, [\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{tid} * \mathcal{R}(\{m\}) \rfloor$$

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 [33,5].

Theorem 1 (Axiom Soundness). *For each axiom, $\Gamma_1 \vdash_{RMO} \alpha \dashv \Gamma_2$, we must show*

$$\forall m.\, [\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{tid} * \mathcal{R}(\{m\}) \rfloor$$

Proof. In Appendix A.1.

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

Lemma 5.

$$\forall \Gamma, C, \Gamma', t.\, \Gamma \vdash_R C \dashv \Gamma' \Rightarrow [\![\Gamma]\!]_t \cap [\![R]\!]_t \vdash [\![C]\!]_t \dashv [\![\Gamma']\!]_t \cap [\![R]\!]_t$$

Proof. In Appendix A.1.

Lemma 6.

$$\forall \Gamma, C, \Gamma', t.\, \Gamma \vdash_M C \dashv \Gamma' \Rightarrow [\![\Gamma]\!]_t \cap [\![M]\!]_t \vdash [\![C]\!]_t \dashv [\![\Gamma']\!]_t \cap [\![M]\!]_t$$

Proof. In Appendix A.1.

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:

$$\forall \alpha, \Gamma_1, \Gamma_2.\, \Gamma_1 \vdash_R \alpha \dashv \Gamma_2 \Rightarrow \\ \forall m.\, [\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{\mathsf{R},tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{\mathsf{R},tid} * \mathcal{R}(\{m\}) \rfloor$$

$$\forall \alpha, \Gamma_1, \Gamma_2.\, \Gamma_1 \vdash_M \alpha \dashv \Gamma_2 \Rightarrow \\ \forall m.\, [\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{\mathsf{M},tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{\mathsf{M},tid} * \mathcal{R}(\{m\}) \rfloor$$

α ranges over any atomic command, such as a field access or variable assignment.

Denoting a type environment $[\![\Gamma]\!]_{\mathsf{M},tid}$, unfolding the definition one step, is merely $[\![\Gamma]\!]_{tid} \cap [\![\mathsf{M}]\!]_{tid}$. 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 7 (Write-Side Critical Section Lifting). *For each α whose semantics does not affect the write lock, if*

$$\forall m.\, [\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{tid} * \mathcal{R}(\{m\}) \rfloor$$

then

$$\forall m.\, [\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{\mathsf{M},tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{\mathsf{M},tid} * \mathcal{R}(\{m\}) \rfloor$$

Proof. In Appendix A.1.

7 Related Work

Related work for correctness of code involving RCU memory management includes relevant work in concurrent program logics, model checking, and type systems.

7.1 Overview on Existing Approaches

Modeling RCU 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 [1,27] satisfies this property. According to their definition, our model in Section 3 is a valid RCU implementation (assuming sequential consistency). To the best of our knowledge, ours is the first abstract *operational* model for a Linux kernel-style RCU implementation — others are implementation-specific [23] or axiomatic like Alglave et al.'s. One another well-known way of implementing RCU synchronization without hurting readers' performance is Quiescent State Based Reclamation(QSBR) [10]. In QSBR model, the writer thread and each reader thread have their own counters.

Program Logics under Sequential Consistency Fu et al. [15] extend Rely-Guarantee/Separation-Logic [37,13,12] with the *past-tense* temporal operator to eliminate the need for using the history variable and lift the standard separation conjuction to assert over on execution histories instead of state to capture the high level intuitive specification of Michael's non-blocking stack [30]. Gotsman et al. [18] take a similar approach and present a formalization of memory management using grace periods in an extension to separation logic, specifically an extension of Vafeiadis and Parkinson's RGSep [37] with assertions from temporal logic to capture the essence of relaxed memory reclamation algorithms to have

simpler proofs than Fu et al. have [15] for Micheal's non-blocking stack [30] implementation under a sequentially consistent memory model. Mandrykin et al. [23] use VCC [7] to verify an implementation of RCU primitives using specifications relating to a specific implementation, and do not verify client code.

Program Logics under Sequential Consistency Tassarotti et al. [34] uses a protocol based program logic based on separation and ghost variables called GPS [36] to verify user-level RCU singly linked list under *release-acquire* semantics, which is a weaker memory model than sequential-consistency. They ensure the safe synchronization in between the readers and the writer thread through *release-writes* and *acquire-reads* of the counters as a part of the QSRB model. We do capture all the guarantees from these *release-acquire* orderings in our modal via *logical observations* – e.g. (iterator *tid*, unlinked and freeable) – the writer thread makes on the heap. Based on comparative understanding of our approach together with Tassarotti's we have the following conclusions:

- Although we do not show it formally within the scope of this paper, we do not see any fundamental reason for having our model to work under *release-acquire* semantics.
- Tassarotti et al. provide *abstract-predicates* – e.g. WriterSafe – (which mainly corresponds to our global memory invariants in Appendix 6.2 and the denotations of our types in Figure 7 Section 6) to be asserted and discharged for each of the RCU client actions and these abstract predicates might not be so reusable for another RCU client – e.g. binary search tree – as they include linked list specific logical constructions. Regarding memory safety of RCU clients, it is important to emphasise that we use the same types for different clients(e.g. a linked list in Appendix C and a binary search tree in Appendix B) without having to prove the RCU properties (e.g. unreachable nodes, proper orderings etc.) separately for each different RCU client: since we introduce the logical observation and memory safety properties within the *meta-theory*(logical RCU state, global memory invariants and the denotations of the types) and prove all of the RCU properties as a part of the soundness proof just for once.

Efficient implementations go to great lengths to minimize the cost of the read-side critical section operations: entering/exiting to the read-side critical section. Typically, implementations are structured so each reader thread (or CPU) maintains its own *fragment* of the shared global state, which is only updated by that thread, but is read by the write-side primitives. For example, Tassarotti et al. model the readers' counters as an array of counters where the content of each cell corresponds to the counter of a specific reader thread. On the other hand, 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. However, we should also note that if we had the careful implementation mentioned for the `ReadBegin` and `ReadEnd` then all actions(traverse and read the node value) from a reader thread would *reduce* to an *atomic* data value-read – the reader threads shows logical atomicity. Since

we do not provide the careful implementation for the the `ReadBegin` and `ReadEnd` we cannot provide the *atomicity* proof as well.

Model Checking Desnoyers et al. propose a virtual architecture to model out of order memory accesses and instruction scheduling [9] and use the SPIN model checker to verify a user-mode implementation of RCU [10]. This method requires manual translation from C to SPIN modelling language. Liang et al. presents an approach to verify the *grace period* guarantee of Tree RCU in Linux kernel [22]. Tree RCU is an hieararchical RCU model which is a remedy to contention of writer threads in the single global lock model. Similarly, Kokologiannakis et al. [20] use a stateless model checker for testing the core of Tree RCU. Liang's approach has limited support for lists and callback handling which are important when we consider the correctness for writer primitives which are based on callback handling. Kokologiannakis's approach overcomes all these limitations and shows considerable performance when compared with Liang's approach using CBMC. Both focus on validating a particular RCU implementation, whereas we focus on verifying memory safety of clients independent of implementation.

Type Systems Howard et al. [19] enhance STM in Haskell to a technique called *Relativistic Programming* which aims low overhead linearly scalable concurrent threads. Unlike our system which types binary search tree in RCU setting, they claim handling trees (look-up followed by update) as a future work. A follow up paper from Cooper et al. [8] presents a Haskell library called *Monadic RP*, for relativistic programming. Similar to our approach, Monadic RP 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. 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.

7.2 More on Efficiency in Real RCU Implementations

[[Iso says: I am no sure this part should be in the ESOP version.]] Our simple global lock semantics for write critical sections has so many drawbacks in terms limiting the scalability and performance. The motivation of RCU is having very cheap reader critical sections but the number of reader theards can be a lot and this can be a problem. Although readers' performance are not effected badly, writer thread must show the grace period to all of these *active/pre-existsting* readers in such a way that it must *defer* the reclamation phase blocking (or in some implementations registering callback to be invoked after grace period) until all active readers are done with their jobs. Acquiring a global lock during each grace period severly affects scalibility and performance. To overcome this problem, a *tree based* hieararchical mechanism is implemented to handle each thread's (CPU's) *quiescent-state* information and *grace-period*. In Tree RCU, the data strcuture is used in such a way that it records each CPU's quiescent states. The rcu node tree propagates these states up to the root, and then propagates grace-period information down to the leaves. Quiescent-state information does not propagate upwards from a given node until a quiescent state has been reported by

each CPU covered by the subtree rooted at that node. This propagation scheme dramatically reduces the lock contention experienced by the upper levels of the tree [22]

Real implementations, of course, must also tolerate the weak memory model of whatever processor they run on, leading them to use. For example, the primitive to dereference a memory location shown by a pointer and the primitive to update the memory location shown by a pointer contain architecture-specific memory barrier instructions and compiler directives to enforce correct ordering. Both primitives reduce to simple assignment statements on sequentially consistent systems. Dereferencing is volatile access except on DEC Alpha, which also requires a memory barrier .

CITE

The Linux implementation of the communication in between reader thread and writer thread is based on waiting for context switches. The implementation uses calls `SyncStart` and returns back from `SyncStop` after all CPUs have passed through at least one context switch. The synchronous version of the primitive to wait reader threads is synchronize_rcu. For performance reasons, the Linux implementation has also asynchronous version of this primitive named call_rcu. Detecting context switches requires maintaining state shared between CPUs. A CPU must update state, which other CPUs read, that indicate it executed a context switch [29].

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. pp. 141–157 (2013). https://doi.org/10.1007/978-3-642-39799-8_9, http://dx.doi.org/10.1007/978-3-642-39799-8_9
2. Alglave, J., Maranget, L., McKenney, P.E., Parri, A., Stern, A.: Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 405–418. ASPLOS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3173162.3177156, http://doi.acm.org/10.1145/3173162.3177156
3. Arbel, M., Attiya, H.: Concurrent updates with rcu: Search tree as an example. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing. pp. 196–205. PODC '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2611462.2611471, http://doi.acm.org/10.1145/2611462.2611471
4. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Proceedings of the 4th International Conference on Formal Methods for Components and Objects. pp. 364–387. FMCO'05, Springer-Verlag, Berlin, Heidelberg (2006). https://doi.org/10.1007/11804192_17, http://dx.doi.org/10.1007/11804192_17
5. 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, 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, http://doi.acm.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 concurrent 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, http://dx.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, http://doi.acm.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, http://doi.acm.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, http://doi.acm.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, http://doi.acm.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, http://doi.acm.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, http://dx.doi.org/10.1134/S0361768816050054
24. Mckenney, P.E.: Exploiting Deferred Destruction: An Analysis of Read-copy-update Techniques in Operating System Kernels. Ph.D. thesis, Oregon Health & Science University (2004), aAI3139819
25. McKenney, P.E.: N4037: Non-transactional implementation of atomic tree move (May 2014), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4037.pdf
26. McKenney, P.E.: Some examples of kernel-hacker informal correctness reasoning. Technical Report paulmck.2015.06.17a (2015), http://www2.rdrop.com/users/paulmck/techreports/IntroRCU.2015.06.17a.pdf
27. Mckenney, P.E.: A tour through rcu's requirements (2017), https://www.kernel.org/doc/Documentation/RCU/Design/Requirements/Requirements.html
28. Mckenney, P.E., Appavoo, J., Kleen, A., Krieger, O., Krieger, O., Russell, R., Sarma, D., Soni, M.: Read-copy update. In: In Ottawa Linux Symposium. pp. 338–367 (2001)
29. Mckenney, P.E., Boyd-wickizer, S.: Rcu usage in the linux kernel: One decade later (Sep 2012), http://rdrop.com/users/paulmck/techreports/survey.2012.09.17a.pdf
30. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (Jun 2004). https://doi.org/10.1109/TPDS.2004.8, http://dx.doi.org/10.1109/TPDS.2004.8
31. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume

9583. pp. 41–62. VMCAI 2016, Springer-Verlag New York, Inc., New York, NY, USA (2016). https://doi.org/10.1007/978-3-662-49122-5_2, http://dx.doi.org/10.1007/978-3-662-49122-5_2

32. Paul E. McKenney, Mathieu Desnoyers, L.J., Triplett, J.: The rcu-barrier menagerie (Nov 2016), https://lwn.net/Articles/573497/
33. 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
34. 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
35. 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
36. 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
37. Vafeiadis, V., Parkinson, M.: A Marriage of Rely/Guarantee and Separation Logic. In: Caires, L., Vasconcelos, V. (eds.) CONCUR 2007 âĂŞ Concurrency Theory, Lecture Notes in Computer Science, vol. 4703, pp. 256–271. Springer Berlin / Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-74407-8_18, 10.1007/978-3-540-74407-8_18

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 5 Section 4.2 and 31 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 $\mathsf{Loc} \to \mathcal{P}(\mathsf{obs})$
- Undefined variable map, U, of type $\mathcal{P}(\mathsf{Var} \times \mathsf{TID})$
- Set of threads, T, of type $\mathcal{P}(\mathsf{TIDS})$
- A to-free map(or free list), F, of type $\mathsf{Loc} \rightharpoonup \mathcal{P}(\mathsf{TID})$

The 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

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

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

$$\mathsf{RCU} = \{o \mid \mathsf{ftype}(f) = \mathsf{RCU} \wedge \exists o'.\, h(o', f) = o\}$$

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 and F)

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

We do our reasoning for soundness over instrumented states and define an erasure relation

$$\lfloor - \rfloor : \mathsf{MState} \implies \mathsf{LState}$$

that projects instrumented states to the common components with MState.

Well-formedness imposes restrictions over the representation of the RCU structure in memory and type of threads for actions defined in operational semantics. For instance, *an unlinked heap location cannot be reached from a root of an RCU data structure*. We define well-formedness as conjuction of memory axioms define in Section A.2 Appendix 6.2.

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

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

Fig. 10: Type Environments

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

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

that yields the set of states corresponding to a given type environment. This is defined in terms of denotation of individual variable assertions

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

The latter is given in Figure 10. To define the former, we first need to state what it means to combine logical machine states.

Composition of instrumented states is an operation

$$\bullet : \mathcal{M} \longrightarrow \mathcal{M} \longrightarrow \mathcal{M}$$

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 11 An important property of composition is that it preserves

$$\bullet = (\bullet_\sigma, \bullet_O, \cup, \cup) \qquad O_1 \bullet_O O_2(loc) \stackrel{\text{def}}{=} O_1(loc) \cup O_2(loc)$$

$$(s_1 \bullet_s s_2) \stackrel{\text{def}}{=} s_1 \cup s_2 \ \texttt{when} \ dom(s_1) \cap dom(s_2) = \emptyset$$

$$(F_1 \bullet_F F_2) \stackrel{\text{def}}{=} F_1 \cup F_2 \ \texttt{when} \ dom(F_1) \cap dom(F_2) = \emptyset$$

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

$$((s,h,l,rt,R,B),O,U,T,F)\mathcal{R}_0((s',h',l',rt',R',B'),O',U',T',F') \stackrel{\text{def}}{=}$$

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

Fig. 11: Composition($\bullet$) and Thread Interference Relation($\mathcal{R}_0$)

validity of logical states:

Lemma 8 (Well Formed Composition). *Any successful composition of two well-formed logical states is well-formed:*

$$\forall_{x,y,z}.\, \mathsf{WellFormed}(x) \implies \mathsf{WellFormed}(y) \implies x \bullet y = z \implies \mathsf{WellFormed}(\mathsf{z})$$

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 $x \bullet y = z$, Wellformed(z) holds. Both x and y have compontents $((s_x, h_x, l_x, rt_x, R_x, B_x), O_x, U_x, T_x, F_x)$ and $((s_y, h_y, l_y, rt_y, R_y, B_y), O_y, U_y, T_y, F_y)$, respectively. $\bullet_s$ operator over stacks s_x and s_y enforces $dom(s_x) \cap dom(s_y) = \emptyset$ 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 $\bullet_F$ 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 $[\![x_1 : T_1, \ldots x_n : T_n]\!]_{tid,\mathsf{R}} = [\![x_1 : T_1]\!]_{tid} \cap \ldots \cap [\![x_n : T_n]\!]_{tid} \cap [\![\mathsf{R}]\!]_{tid}$ where $[\![\mathsf{R}]\!]_{tid} = \{(s, h, l, rt, R, B), O, U, T, F \mid tid \in R\}$
- For write-side as $[\![x_1 : T_1, \ldots x_n : T_n]\!]_{tid,\mathsf{M}} = [\![x_1 : T_1]\!]_{tid} \cap \ldots \cap [\![x_n : T_n]\!]_{tid} \cap [\![\mathsf{M}]\!]_{tid}$ where $[\![\mathsf{M}]\!]_{tid} = \{(s, h, l, rt, R, B), O, U, T, F \mid tid = l\}$
- $[\![x_1 : T_1, \ldots x_n : T_n]\!]_{tid,\mathsf{O}} = [\![x_1 : T_1]\!]_{tid} \cap \ldots \cap [\![x_n : T_n]\!]_{tid} \cap [\![\mathsf{O}]\!]_{tid}$ where $[\![\mathsf{O}]\!]_{tid} = \{(s, h, l, rt, R, B), O, U, T, F \mid tid \neq l \wedge tid \notin R\}$.

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

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

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 $\mathcal{R}$ on $\mathcal{M}$, and a View in the formal framework is then:

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

Thread interference relation

$$\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$$

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

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

where $\mathcal{R}$ is transitive-reflexive closure of $\mathcal{R}_0$ shown at Figure 11. $\mathcal{R}_0$ (and therefore $\mathcal{R}$) also "preserves" validity:

Lemma 9 (Valid $\mathcal{R}_0$ Interference). *For any m and m', if* $\mathsf{WellFormed}(m)$ *and $m\mathcal{R}_0 m'$, then* $\mathsf{WellFormed}(m')$.

Proof. By assumption, we know that $m = (s, h, l, rt, R, B), O, U, T, F)$ is wellformed. 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

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

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

and by assumptions from the lemma($\mathsf{WellFormed}(m)$.**RINFL**) we can conclude that F,l and h do not have effect on wellformedness of the m.

Lemma 10 (Stable Environment Denotation-M). *For any* closed *environment* Γ *(i.e.,* $\forall x \in \mathsf{dom}(\Gamma).\, \mathsf{FV}(\Gamma(x)) \subseteq \mathsf{dom}(\Gamma)$*):*

$$\mathcal{R}(\llbracket \Gamma \rrbracket_{\mathsf{M},tid}) \subseteq \llbracket \Gamma \rrbracket_{\mathsf{M},tid}$$

Alternatively, we say that environment denotation is stable *(closed under* $\mathcal{R}$*).*

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

$$\llbracket \Gamma' \rrbracket_{\mathsf{M},tid}$$

is stable, and must show that

$$\llbracket \Gamma' \rrbracket_{\mathsf{M},tid} \cap \llbracket x : \tau \rrbracket_{tid}$$

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 $tid = l$. So we can derive conclusion for preservation of F and h and l by

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

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

Cases in which denotations, $\llbracket x : T \rrbracket$, touching these R_0 *preserved* maps are trivial to show.

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

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

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

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

prove this case.

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

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

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

prove this case.

Lemma 11 (Stable Environment Denotation-R). *For any* closed *environment* Γ *(i.e.,* $\forall x \in \mathsf{dom}(\Gamma).\, , \mathsf{FV}(\Gamma(x)) \subseteq \mathsf{dom}(\Gamma)$*):*

$$\mathcal{R}(\llbracket \Gamma \rrbracket_{\mathsf{R},tid}) \subseteq \llbracket \Gamma \rrbracket_{\mathsf{R},tid}$$

Alternatively, we say that environment denotation is stable *(closed under* $\mathcal{R}$*).*

Proof. Proof is similar to one for Lemma 10 where there is only one simple case, $\llbracket x : \mathsf{rcultr} \rrbracket$.

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:

$$\forall \Gamma, C, \Gamma', tid.\, \Gamma \vdash_{M,R} C \dashv \Gamma' \Rightarrow \{\llbracket \Gamma \rrbracket_{tid}\} \llbracket C \rrbracket_{tid} \{\llbracket \Gamma' \rrbracket_{tid}\}$$

The antecedent of the implication is a type judgment shown in Figures 5 Section 4.3, 4 Section 4.1 and 31 Appendix E, and the conclusion is a judgment in the Views logic. The environments are translated to views ($\mathsf{View}_{\mathcal{M}}$) 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 12. $\mathsf{assume}(b)$ is a standard construct in verification semantics [4] [31], which "does nothing" (freezes) if the condition b is false, so its postcondition in the Views logic can reflect the truth of b. 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 α ensuring that the operation interpretation of the action satisfies the specification:$p \sqsubseteq q \overset{def}{\Leftrightarrow} \forall m \in \mathcal{M}.\, \lfloor p * \{m\} \rfloor \subseteq \lfloor q * \mathcal{R}(\{m\}) \rfloor$. Because the Views framework handles soundness for the structural rules (sequencing, parallel composition, etc.), there are really

$$[\![\mathsf{if}\ (x.f == y)\ C_1\ C_2]\!]_{tid} = z = x.f; ((\mathsf{assume}(z = y); C_1) + (\mathsf{assume}(z \neq y); C_2))$$

$$[\![\mathtt{assume}(\mathcal{S})]\!](s) \stackrel{\text{def}}{=} \begin{cases} \{s\} \text{ if } s \in \mathcal{S} \\ \emptyset \quad \text{Otherwise} \end{cases} \quad \frac{\{P\} \cap \{\lceil \mathcal{S} \rceil\} \sqsubseteq \{Q\}}{\{P\}\mathtt{assume}\,(b)\,\{Q\}}\ \mathsf{where}$$

$$\lceil \mathcal{S} \rceil = \{m | \lfloor m \rfloor \cap \mathcal{S} \neq \emptyset\}\ [\![\mathsf{while}\ (e)\ C]\!]_{tid} = (\mathsf{assume}(e); C)^*; (\mathsf{assume}(\neg e));$$

Fig. 12: Encoding of assume(b)

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 $\Gamma <: \Gamma'$, then $[\![\Gamma]\!] \sqsubseteq [\![\Gamma']\!]$. And finally, we must prove that each atomic action's type rule corresponds to a valid semantic judgment in the Views Framework:

$$\forall m.\, [\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{tid} * \mathcal{R}(\{m\}) \rfloor$$

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 [33,5].

Theorem 2 (Axiom Soundness). *For each axiom, $\Gamma_1 \vdash_{RMO} \alpha \dashv \Gamma_2$, we must show*

$$\forall m.\, [\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{tid} * \mathcal{R}(\{m\}) \rfloor$$

Proof. By case analysis on the atomic action α followed by inversion on typing derivation.

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

Lemma 12.

$$\forall \Gamma, C, \Gamma', t.\, \Gamma \vdash_R C \dashv \Gamma' \Rightarrow [\![\Gamma]\!]_t \cap [\![R]\!]_t \vdash [\![C]\!]_t \dashv [\![\Gamma']\!]_t \cap [\![R]\!]_t$$

Proof. Proof is similar to the one for Lemma 13 except the denotation for type system definition is $[\![R]\!]_t = \{\{((s, h, l, rt, R, B), O, U, T, F) | t \in R\}$ which shrinks down the set of all logical states to the one that can only be defined by types(rcultr) in read type system.

Lemma 13.

$$\forall \Gamma, C, \Gamma', t.\, \Gamma \vdash_M C \dashv \Gamma' \Rightarrow [\![\Gamma]\!]_t \cap [\![M]\!]_t \vdash [\![C]\!]_t \dashv [\![\Gamma']\!]_t \cap [\![M]\!]_t$$

Proof. Induction on derivation of $\Gamma \vdash_M C \dashv \Gamma'$ and then inducting on the type of first element of the environment. For the nonempty case, $\Gamma'', x : T$ we do case analysis on T. Type environment for write-side actions includes only: rcultr $\rho\mathcal{N}$, undef, rcuFresh, unlinked and 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:

$$\begin{array}{l}\forall \alpha, \Gamma_1, \Gamma_2. \, \Gamma_1 \vdash_R \alpha \dashv \Gamma_2 \Rightarrow \\ \quad \forall m. \, [\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{\mathsf{R},tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{\mathsf{R},tid} * \mathcal{R}(\{m\}) \rfloor\end{array}$$

$$\begin{array}{l}\forall \alpha, \Gamma_1, \Gamma_2. \, \Gamma_1 \vdash_M \alpha \dashv \Gamma_2 \Rightarrow \\ \quad \forall m. \, [\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{\mathsf{M},tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{\mathsf{M},tid} * \mathcal{R}(\{m\}) \rfloor\end{array}$$

α ranges over any atomic command, such as a field access or variable assignment.

Denoting a type environment $[\![\Gamma]\!]_{\mathsf{M},tid}$, unfolding the definition one step, is merely $[\![\Gamma]\!]_{tid} \cap [\![\mathsf{M}]\!]_{tid}$. 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 14 (Write-Side Critical Section Lifting). *For each α whose semantics does not affect the write lock, if*

$$\forall m. \, [\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{tid} * \mathcal{R}(\{m\}) \rfloor$$

then

$$\forall m. \, [\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{M,tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{M,tid} * \mathcal{R}(\{m\}) \rfloor$$

Proof. Each of these shared actions α preserves the lock component of the physical state, the only component constrained by $[\![-]\!]_{M,tid}$ beyond $[\![-]\!]_{tid}$. For the read case, we must prove from the assumed subset relationship that for an aritrary m:

$$[\![\alpha]\!](\lfloor [\![\Gamma_1]\!]_{tid} \cap [\![\mathsf{M}]\!]_{tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma_2]\!]_{tid} \cap [\![\mathsf{M}]\!]_{tid} * \mathcal{R}(\{m\}) \rfloor$$

By assumption, transitivity of $\subseteq$, and the semantics for the possible αs, the left side of this containment is already a subset of

$$\lfloor [\![\Gamma_2]\!]_{tid} * \mathcal{R}(\{m\}) \rfloor$$

What remains is to show that the intersection with $[\![\mathsf{M}]\!]_{tid}$ is preserved by the atomic action. This follows from the fact that none of the possible αs modifies the global lock.

$$\mathbf{OW}(\sigma, O, U, T, F) = \begin{cases} \forall_{o,o'f,f'}.\, \sigma.h(o,f) = v \wedge \sigma.h(o',f') = v \\ \wedge v \in \mathsf{OID} \wedge \mathsf{FType}(f) = \mathsf{RCU} \implies \\ \quad \begin{cases} o = o' \wedge f = f' \\ \vee \mathsf{unlinked} \in O(o) \\ \vee \mathsf{unlinked} \in O(o') \\ \vee \mathsf{freeable} \in O(o) \\ \vee \mathsf{freeable} \in O(o') \\ \vee \mathsf{fresh} \in O(o)) \\ \vee \mathsf{fresh} \in O(o') \end{cases} \end{cases}$$

Fig. 13: Ownership

$$\mathbf{RWOW}(\sigma, O, U, T, F) = \begin{cases} \forall x, tid, o.\, \sigma.s(x, tid) = o \wedge (x, tid) \notin U \implies \\ \quad \begin{cases} \mathsf{iterator}\ \ tid \in O(o) \\ \vee(\sigma.l = tid \wedge (\mathsf{unlinked} \in O(o)) \\ \vee(\sigma.l = tid \wedge \mathsf{freeable} \in O(o))) \\ \vee(\sigma.l = tid \wedge \mathsf{fresh} \in O(o)) \end{cases} \end{cases}$$

Fig. 14: Reader-Writer-Iterator-Coexistence-Ownership

A.2 Complete Memory Axioms

1. Ownership invariant in Figure 13 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 14 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 15 asserts that the unique root location can only be aliased with thread local references through which the unique root location is observed as iterator.

$$\mathbf{AWRT}(\sigma, O, U, T, F) = \{(\forall_{y,tid}.\, h^*(\sigma.rt, \epsilon) = s(y, tid) \implies \mathsf{iterator}\, tid \in O(s(y, tid)))$$

Fig. 15: Alias with Unique Root

4. Iterators-Free-List invariant in Figure 16 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.

$$\mathbf{IFL}(\sigma, O, U, T, F) = \left\{ \forall tid, o.\, \mathsf{iterator}\, tid \in O(o) \wedge \forall_{T' \subseteq T}.\, \sigma.F([o \mapsto T']) \implies tid \in T' \right.$$

Fig. 16: Iterators-Free-List

5. Unlinked-Reachability invariant in Figure 17 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.

$$\mathbf{ULKR}(\sigma, O, U, T, F) = \left\{ \begin{array}{l} \forall o.\, \mathsf{unlinked} \in O(o) \implies \\ \left\{ \begin{array}{l} \forall o', f'.\, \sigma.h(o', f') = o \implies \\ \quad \left\{ \begin{array}{l} \mathsf{unlinked} \in O(o') \vee \\ \mathsf{freeable} \in O(o') \end{array} \right. \end{array} \right. \end{array} \right.$$

Fig. 17: Unlinked-Reachability

6. Free-List-Reachability invariant in Figure 18 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.

$$\mathbf{FLR}(\sigma, O, U, T, F) = \left\{ \begin{array}{l} \forall o.\, F([o \mapsto T]) \implies \\ \left\{ \begin{array}{l} \forall o', f'.\, \sigma.h(o', f') = o \implies \\ \quad \left\{ \exists_{T' \subseteq T}.\, F([o' \mapsto T']) \right. \end{array} \right. \end{array} \right.$$

Fig. 18: Free-List-Reachability

7. Writer-Unlink invariant in Figure 19 asserts that the writer thread cannot observe a heap location as unlinked.

$$\mathbf{WULK}(\sigma, O, U, T, F) = \left\{ \forall o.\, \mathsf{iterator}\, \sigma.l \in O(o) \implies \mathsf{unlinked} \notin O(o) \right.$$

Fig. 19: Writer-Unlink

8. Fresh-Reachable invariant in Figure 20 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.

$$\mathbf{FR}(\sigma, O, U, T, F) = \begin{array}{l}\forall_{tid,x,o}.\,(\sigma.s(x, tid) = o \wedge \mathsf{fresh} \in O(o)) \implies \\ (\forall_{y,o',f',tid'}.(h(o', f') \neq o) \vee (s(y, tid) \neq) o \vee (tid' \neq tid \implies s(y, tid') \neq o))\end{array}$$

Fig. 20: Fresh-Reachable

9. Fresh-Writer invariant in Figure 21 asserts that heap allocation can be done only by writer thread.

$$\mathbf{WF}(\sigma, O, U, T, F) = \forall_{tid,x,o}.\,(\sigma.s(x, tid) = o \wedge \mathsf{fresh} \in O(o)) \implies tid = \sigma.l$$

Fig. 21: Fresh-Writer

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

$$\mathbf{FNR}(\sigma, O, U, T, F) = \forall_o.\,(\mathsf{fresh} \in O(o)) \implies (\forall_{x,tid}.\,\mathsf{iterator}\,tid \notin O(o)) \wedge \mathsf{unlinked} \notin O(o)$$

Fig. 22: Fresh-Not-Reader

11. Fresh-Points-Iterator invariant in Figure 23 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 $\mathcal{N} = \mathcal{N}'$ asserted in the type rule for fresh node linking(T-LinkF).

$$\mathbf{FPI}(\sigma, O, U, T, F) = \forall_o.\,(\mathsf{fresh} \in O(o) \wedge \exists_{f,o'}.\,h(o, f) = o') \implies (\forall_{tid}.\,\mathsf{iterator}\,tid \in O(o'))$$

Fig. 23: Fresh-Points-Iterator

12. Writer-Not-Reader invariant in Figure 24 asserts that a writer thread identifier can not be a reader thread identifier.
13. Readers-Iterator-Only invariant in the Figure 25 asserts that a reader threads can only make iterator observation on a heap location.
14. Readers-In-Free-List invariant in Figure 26 asserts that for any mapping from a location to a set of threads in the free list we know the fact that this set

$$\mathbf{WNR}(\sigma, O, U, T, F) = \left\{ \sigma.l \notin \sigma.R \right.$$

Fig. 24: Writer-Not-Reader

$$\mathbf{RITR}(\sigma, O, U, T, F) = \left\{ \forall_{tid \in \sigma.R, o}.\, \mathsf{iterator}\, tid \in O(o) \right.$$

Fig. 25: Readers-Iterator-Only

of threads is a subset of bounding threads(which itself is subset of reader threads).

$$\mathbf{RINFL}(\sigma, O, U, T, F) = \left\{ \forall_o.\, F([o \mapsto T]) \implies T \subseteq \sigma.B \right.$$

Fig. 26: Readers-In-Free-List

15. Heap-Domain invariant in the Figure 27 defines the domain of the heap.

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

Fig. 27: Heap-Domain

16. Unique-Root invariant in Figure 28 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*.

$$\mathbf{UNQRT}(\sigma, O, U, T, F) = \left\{ \forall_{\rho \neq \epsilon}.\, \mathsf{iterator}\, tid \in O(h^*(\sigma.rt, \rho) \wedge \neg(\exists_{f'}.\, \sigma.rt = h(h^*(\sigma.rt, \rho), f')) \right.$$

Fig. 28: Unique-Root

17. Unique-Path invariant in Figure 29 asserts that every node is reachable from root node with an unique path. This invariant is a part of acyclicity enforcement on the heap layout of the data structure.

$$\mathbf{UNQR}(\sigma, O, U, T, F) = \left\{ \forall_{\rho,\rho'}.\, h^*(\sigma.rt, \rho) \neq h^*(\sigma.rt, \rho') \implies \rho \neq \rho' \right.$$

Fig. 29: Unique-Reachable

Each of these memory invariants captures different aspects of validity of the memory under RCU setting, $\mathsf{WellFormed}(\sigma, O, U, T, F)$, 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 15 (Unlink).

$$[\![x.f_1 := r]\!](\lfloor [\![\Gamma,\, x : \mathsf{rcultr}\,\rho\,\mathcal{N}([f_1 \rightharpoonup z]), z : \mathsf{rcultr}\,\rho'\,\mathcal{N}'([f_2 \rightharpoonup r]),\; r : \mathsf{rcultr}\,\rho''\,\mathcal{N}'']\!]_{M,tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma,\, x : \mathsf{rcultr}\,\rho\,\mathcal{N}(f_1 \rightharpoonup z \setminus r),\; z : \mathsf{unlinked},\; r : \mathsf{rcultr}\,\rho'\,\mathcal{N}'']\!] * \mathcal{R}(\{m\}) \rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F) \in [\![\Gamma,\, x : \mathsf{rcultr}\,\rho\,\mathcal{N},\; z : \mathsf{rcultr}\,\rho'\,\mathcal{N}',\; r : \mathsf{rcultr}\,\rho''\,\mathcal{N}'']\!]_{M,tid} * \{m\} \tag{1}$$

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

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

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

$$\forall_{f \in dom(\mathcal{N}')}.\, f \neq f_2 \implies \mathcal{N}'(f) = \mathsf{null} \tag{4}$$

$$\forall_{n \in \Gamma, m, \mathcal{N}''', p''', f}.\, n : \mathsf{rcultr}\,\rho'''\,\mathcal{N}'''([f \rightharpoonup m]) \implies \begin{cases} ((\neg \mathsf{MayAlias}(\rho''', \{\rho, \rho', \rho''\})) \wedge (m \notin \{z, r\})) \\ \wedge (\forall_{\rho'''' \neq \epsilon}.\, \neg \mathsf{MayAlias}(\rho''', \rho''.\rho'''')) \end{cases} \tag{5}$$

We split the composition in 1 as

$$(\sigma_1, O_1, U_1, T_1, F_1) \in [\![\Gamma,\, x : \mathsf{rcultr}\,\rho\,\mathcal{N},\; z : \mathsf{rcultr}\,\rho'\,\mathcal{N}',\; r : \mathsf{rcultr}\,\rho''\,\mathcal{N}'']\!]_{M,tid} \tag{6}$$

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

$$\sigma_1 \bullet_s \sigma_2 = \sigma \tag{8}$$

$$O_1 \bullet_O O_2 = O \tag{9}$$

$$U_1 \cup U_2 = U \tag{10}$$

$$T_1 \cup T_2 = T \tag{11}$$

$$F_1 \uplus F_2 = F \tag{12}$$

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

$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{14}$$

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

$$(\sigma_1', O_1', U_1', T_1', F_1') \in [\![\Gamma,\, x : \mathsf{rcultr}\,\rho\mathcal{N}([f_1 \rightharpoonup r]),\, z : \mathsf{unlinked}, r : \mathsf{rcultr}\,\rho'\,\mathcal{N}'']\!]_{M,tid} \tag{15}$$

$$\tag{16}$$

$$\mathcal{N}(f_1) = r \tag{17}$$

$$(\sigma_2', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{18}$$

$$\sigma_1' \bullet_s \sigma_2' = \sigma' \tag{19}$$

$$O_1' \bullet_O O_2' = O' \tag{20}$$

$$U_1' \cup U_2' = U' \tag{21}$$

$$T_1' \cup T_2' = T' \tag{22}$$

$$F_1' \uplus F_2' = F' \tag{23}$$

$$\mathsf{WellFormed}(\sigma_1', O_1', U_1', T_1', F_1') \tag{24}$$

$$\mathsf{WellFormed}(\sigma_2', O_2', U_2', T_2', F_2') \tag{25}$$

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

$$\sigma_1' = \sigma_1[h(s(x, tid), f_1) \mapsto s(r, tid)] \tag{26}$$

and 26 is determined by operational semantics.

The only change in the observation map is on $s(y, tid)$ from $\mathsf{iterator}\ tid$ to $\mathsf{unliked}$

$$O_1' = O_1(s(y, tid))[\mathsf{iterator}\ tid \mapsto \mathsf{unlinked}] \tag{27}$$

28 follows from 1

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{28}$$

σ_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, T_1' as T_1 and F_1' as F_1. Assuming 6 and choices on maps makes $(\sigma_1', O_1', U_1', T_1')$ in denotation

$$[\![\Gamma,\, x : \mathsf{rcultr}\,\rho\mathcal{N}([f \rightharpoonup r]), z : \mathsf{unlinked}, r : \mathsf{rcultr}\,\rho'\,\mathcal{N}'']\!]_{M,tid}$$

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(3-5), denotations of the precondition(6) and 13.**UNQR**

$$\rho \neq \rho' \neq \rho'' \tag{29}$$

and

$$o_x \neq o_y \neq o_z \tag{30}$$

where o_x, o_y and o_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(o_x, o_y, o_z and ρ, ρ') are unique.

We must prove

$$h'^*(\sigma.rt, \rho) \neq h'^*(\sigma.rt, \rho.f_1) \implies \rho \neq \rho.f_1 \tag{31}$$

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 ρ and $\rho.f_1$ are observed as $\mathsf{iterator}\, tid$ including the final reached heap locations($\mathsf{iterator}\, tid \in O'_1(\sigma'.h^*(\sigma.rt, \rho))$ and $\mathsf{iterator}\, tid \in O'_1(\sigma'.h^*(\sigma.rt, \rho.f_1))$). 17 is determined directly by operational semantics.

$\mathsf{unlinked} \in O'_1(o_y)$ follows from 27 and 26 which makes path $\rho.f_1.f_2$ invalid(from denotation(15), all heap locations reaching to $O'_1(o_r)$ from root($\sigma.rt$) are observed as $\mathsf{iterator}\, tid$ so this proves that $\mathsf{unlinked} \in O'_1(o_y)$) 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

$$\rho \neq \rho' \tag{32}$$

$$o_x \neq o_y \neq o_z \tag{33}$$

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 15, 27 and 28.

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

Case 11. - **AWRT** By 15.

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

Case 13. - **WNR** By 13.**WNR**, 26, 27 and 28.

Case 14. - **RINFL** By 15, 13.**RINFL**, choice of F'_1 and 26.

Case 15. - **ULKR** We must prove 34

$$\forall_{o',f'}.\, \sigma'.h(o',f') = o_y \implies \mathsf{unlinked} \in O_1'(o') \\ \vee (\mathsf{freeable} \in O_1'(o')) \tag{34}$$

which follows from 15, 13.**OW**, operational semantics(26) and 27. If o' were observed as iterator then that would conflict with 24.**UNQR**.

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

To prove 18 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{35}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{36}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{37}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{38}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{39}$$

$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{40}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h) \tag{41}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h') \tag{42}$$

To prove all relations (35-40) we assume 28 which is to assume T_2 as subset of reader threads. Let σ_2' be σ_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 36 follow from 28 and choice of F_2'. 41, 42 and 39 are determined by choice of σ_2', operational semantic and choices made on maps related to the assertions.

By assuming 14 we show 25. 37 and 38 follow trivially. 40 follows from choice of σ_2', 26 and 28.

To prove 20 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

$$\mathsf{iterator}\, tid \in O_2'(o_y)$$

We also know from 27 that

$$\mathsf{unliked} \in O_1'(o_y)$$

Both together with 9 and 15 proves 20.

To show 19 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 24.**OW-HD** and 25.**OW-HD**. 21, 22 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 16 (LinkFresh).

$$[\![p.f := n]\!](\lfloor [\![\Gamma,\, p : \mathsf{rcultr}\,\rho\,\mathcal{N}\,, r : \mathsf{rcultr}\,\rho'\,\mathcal{N}'\,, n : \mathsf{rcuFresh}\,\mathcal{N}'']\!]_{M,tid} * \{m\}\rfloor) \subseteq$$
$$\lfloor [\![\Gamma\,, p : \mathsf{rcultr}\,\rho\,\mathcal{N}([f \rightharpoonup r \setminus n])\,, n : \mathsf{rcultr}\,\rho'\,\mathcal{N}''\,, r : \mathsf{unlinked}]\!] * \mathcal{R}(\{m\})\rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F) \in [\![\Gamma,\, p : \mathsf{rcultr}\,\rho\,\mathcal{N}\,, r : \mathsf{rcultr}\,\rho'\,\mathcal{N}'\,, n : \mathsf{rcuFresh}\,\mathcal{N}'']\!]_{M,tid} * \{m\} \tag{43}$$

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{44}$$

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

$$\mathsf{FV}(\Gamma) \cap \{p, r, n\} = \emptyset \tag{45}$$
$$\rho.f = \rho' \text{ and } \mathcal{N}(f) = r \tag{46}$$
$$\mathcal{N}' = \mathcal{N}'' \tag{47}$$
$$\forall_{x \in \Gamma, \mathcal{N}''', \rho'', f', y}.\, (x : \mathsf{rcultr}\,\rho''\,\mathcal{N}'''([f' \rightharpoonup y])) \implies (\neg \mathsf{MayAlias}(\rho'', \{\rho, \rho'\}) \wedge (y \neq o)) \tag{48}$$

We split the composition in 43 as

$$(\sigma_1, O_1, U_1, T_1, F_1) \in [\![\Gamma,\, p : \mathsf{rcultr}\,\rho\,\mathcal{N}\,, r : \mathsf{rcultr}\,\rho'\,\mathcal{N}'\,, n : \mathsf{rcuFresh}\,\mathcal{N}'']\!]_{M,tid} \tag{49}$$

$$(\sigma_2, O_2, U_2, T_2, F_2) = m \tag{50}$$
$$O_1 \bullet_O O_2 = O \tag{51}$$
$$\sigma_1 \bullet_s \sigma_2 = \sigma \tag{52}$$
$$U_1 \cup U_2 = U \tag{53}$$
$$T_1 \cup T_2 = T \tag{54}$$
$$F_1 \uplus F_2 = F \tag{55}$$
$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1) \tag{56}$$
$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{57}$$

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

$$(\sigma_1', O_1', U_1', T_1', F_1') \in [\![p : \mathsf{rcultr}\,\rho\,\mathcal{N}\,, n : \mathsf{rcultr}\,\rho'\,\mathcal{N}''\,, r : \mathsf{unlinked}\,, \Gamma]\!]_{M,tid} \tag{58}$$
$$\mathcal{N}(f) = n \tag{59}$$
$$(\sigma_2', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{60}$$
$$O_1' \bullet_O O_2' = O' \tag{61}$$
$$\sigma_1' \bullet_s \sigma_2' = \sigma' \tag{62}$$
$$U_1' \cup U_2' = U' \tag{63}$$
$$T_1' \cup T_2' = T' \tag{64}$$
$$F_1' \uplus F_2' = F' \tag{65}$$
$$\mathsf{WellFormed}(\sigma_1', O_1', U_1', T_1', F_1') \tag{66}$$
$$\mathsf{WellFormed}(\sigma_2', O_2', U_2', T_2', F_2') \tag{67}$$

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

$$\sigma_1' = \sigma_1[h(s(p, tid), f) \mapsto s(n, tid)] \tag{68}$$

59 is determined directly from operational semantics.

We know that changes in observation map are

$$O_1' = O_1(s(r, tid))[\mathsf{iterator}\ tid \mapsto \mathsf{unlinked}] \tag{69}$$

and

$$O_1' = O_1(s(n, tid))[\mathsf{fresh} \mapsto \mathsf{iterator}\, tid] \tag{70}$$

71 follows from 43

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{71}$$

Let T_1' be T_1, F_1' be F_1 and σ_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

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

In the rest of the proof, we prove 66, 67 and show the composition of $(\sigma_1', O_1', U_1', T_1')$ and $(\sigma_2', O_2', U_2', T_2')$. To prove 66, we need to show that each of the memory axioms in Section A.2 holds for the state $(\sigma', O_1', U_1', T_1')$.

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(49), 13.**FR** and 56.**UNQR**

$$\rho \neq \rho.f \neq \forall_{\mathcal{N}'([f_i \rightharpoonup x_i])}.\, \rho.f.f_i \tag{72}$$

and

$$o_p \neq o_r \neq o_n \neq o_i \text{ where } o_i = h(o_r, f_i) \tag{73}$$

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 56.**FR**, we assume that there exists no field alias/path alias to heap location freshly allocated o_n).

We must prove

$$\rho \neq \rho.f \neq \rho.f.f_i \iff \sigma'.h^*(\sigma.rt, \rho) \neq \sigma'.h^*(\sigma.rt, \rho.f)) \neq \sigma'.h^*(\sigma.rt, \rho.f.f_i)) \tag{74}$$

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 ρ and $\rho.f$ are observed as $\mathsf{iteartor}\, tid$ including the final reached heap

locations($\mathsf{iterator}\, tid \in O_1'(\sigma'.h^*(\sigma.rt, \rho))$, $\mathsf{iterator}\, tid \in O_1'(\sigma'.h^*(\sigma.rt, \rho.f))$ and $\mathsf{iterator}\, tid \in O_1'(\sigma'.h^*(\sigma.rt, \rho.f.f_i))$). The preservation of uniqueness follows from 69, 70, 68 and 56.**FR**.

from which we conclude 75 and 76

$$\rho \neq \rho.f \neq \rho.f.f_i \tag{75}$$

$$o_p \neq o_n \neq o_r \tag{76}$$

from which 74 follows.

Case 18. - **OW** By 56.**OW**, 68, 69 and 70.

Case 19. - **RWOW** By 56.**RWOW**, 68, 69 and 70

Case 20. - **AWRT** Trivial.

Case 21. - **IFL** By 56.**WULK**, 69, 70 choice of F_1' and operational semantics.

Case 22. - **FLR** By choice of F_1' and 56.

Case 23. - **FPI** By 58.

Case 24. - **WULK** Determined by operational semantics By 56.**WULK**, 69, 70 and operational semantics.

Case 25. - **WF** and **FR** Trivial.

Case 26. - **HD**

Case 27. - **WNR** By 71 and operational semantics.

Case 28. - **RINFL** Determined by operational semantics(68) and 56.**RINFL**.

Case 29. - **ULKR** We must prove

$$\forall_{o',f'}.\, \sigma'.h(o', f') = o_r \implies \begin{array}{l} \mathsf{unlinked} \in O_1'(o') \\ \mathsf{freeable} \in O_1'(o') \end{array} \tag{77}$$

which follows from 58, 56.**OW** and determined by operational semantics(68), 69, 70. If o' were observed as $\mathsf{iterator}$ then that would conflict with 66.**UNQR**.

Case 30. - **UNQRT** By 56.**UNQRT**, 69, 70 and 68.

To prove 60, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{78}$$
$$l \in T_2 \rightarrow F_2 = F_2' \tag{79}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{80}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{81}$$
$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{82}$$
$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{83}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h) \tag{84}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h') \tag{85}$$

To prove all relations (78-83) we assume 71 which is to assume T_2 as subset of reader threads. Let σ_2' be σ_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. 83 follows from choice of σ_2', 68 and 71.

78 and 79 follow from 71 and choice of F_2'. 82, 84 and 85 are determined by choice of σ_2', operational semantics and choices made on maps related to the assertions.

To prove 61 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

$$\mathsf{iterator}\, tid \in O_2'(o_r) \tag{86}$$

From 69 we know that

$$\mathsf{unliked} \in O_1'(o_r)$$

Both together with 51 and 58 proves 61.

For case 87

$$\mathsf{fresh} \in O_2(o_n) \tag{87}$$

From 70 we know that

$$\mathsf{iterator}\, tid \in O_1'(o_n)$$

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-HD** and 67.**OW-HD**. 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 $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$.

Lemma 17 (ReadStack).

$$[\![z := x]\!](\lfloor [\![\Gamma, z : \_, x : \mathit{rcultr}\ \rho\ \mathcal{N}]\!]_{M,tid} * \{m\}\rfloor) \subseteq$$
$$\lfloor [\![\Gamma, x : \mathit{rcultr}\ \rho\ \mathcal{N}, z : \mathit{rcultr}\ \rho\ \mathcal{N}]\!] * \mathcal{R}(\{m\})\rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F) \in [\![\Gamma, z : \_, x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} * \{m\} \tag{88}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{89}$$

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

$$\mathsf{FV}(\Gamma) \cap \{z\} = \emptyset \tag{90}$$

We split the composition in 88 as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![\Gamma, z : \_, x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} \tag{91}$$
$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{92}$$
$$\sigma_1 \bullet \sigma_2 = \sigma \tag{93}$$
$$O_1 \bullet_O O_2 = O \tag{94}$$
$$U_1 \cup U_2 = U \tag{95}$$
$$T_1 \cup T_2 = T \tag{96}$$
$$F_1 \uplus F_2 = F \tag{97}$$
$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1) \tag{98}$$
$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{99}$$

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

$$(\sigma', O_1', U_1', T_1', F_1') \in [\![\Gamma, x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}, z : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} \tag{100}$$
$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{101}$$
$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{102}$$
$$O_1' \bullet_O O_2' = O' \tag{103}$$
$$U_1' \cup U_2' = U' \tag{104}$$
$$T_1' \cup T_2' = T' \tag{105}$$
$$F_1' \uplus F_2' = F' \tag{106}$$
$$\mathsf{WellFormed}(\sigma_1', O_1', U_1', T_1', F_1') \tag{107}$$
$$\mathsf{WellFormed}(\sigma_2', O_2', U_2', T_2', F_2') \tag{108}$$

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

$$\sigma' = \sigma[s(z, tid) \mapsto o_x] \tag{109}$$

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

$$O_1' = O_1 \tag{110}$$

111 follows from 88

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{111}$$

σ_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 91 and choices on maps makes $(\sigma_1', O_1', U_1', T_1')$ in denotation

$$[\![\Gamma\,, x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}\,, z : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid}$$

In the rest of the proof, we prove 107, 108 and show the composition of $(\sigma_1', O_1', U_1', T_1', F_1')$ and $(\sigma_2', O_2', U_2', T_2', F_2')$. 107 follows from 98 trivially.

To prove 101, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{112}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{113}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{114}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{115}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R_2 = \sigma_2'.R_2 \wedge \sigma_2.rt = \sigma_2'.rt \tag{116}$$

$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{117}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h) \tag{118}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h') \tag{119}$$

To prove all relations (112-117) we assume 111 which is to assume T_2 as subset of reader threads. Let σ_2' be σ_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 99 we show 108. 114, 115, 118 and 119 follow trivially. 117 follows from choice of σ_2' and 109(determined by operational semantics).

112 and 113 follow from 111 and choice of F_2'. 116, 118 and 119 are determined by choice of σ_2', operational semantics and choices made on maps related to the assertions.

103-106 are trivial by choices on related maps and semantics of the composition operators for these maps. 102 follows trivially from 93. All compositions shown let us to derive conclusion for $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$ trivial.

Lemma 18 (ReadHeap).

$$[\![z := x.f]\!](\lfloor [\![\Gamma\,, z : \_\,, x : \mathit{rcultr}\,\rho\,\mathcal{N}]\!]_{M,tid} * \{m\}\rfloor) \subseteq$$
$$\lfloor [\![\Gamma\,, x : \mathit{rcultr}\,\rho\,\mathcal{N}[f \mapsto z]\,, z : \mathit{rcultr}\,\rho'\,\mathcal{N}_\emptyset]\!] * \mathcal{R}(\{m\})\rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F)\ \in [\![\Gamma\,, z : \mathsf{rcultr}\,\_\,, x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} * \{m\} \tag{120}$$

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{121}$$

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

$$\mathsf{FV}(\Gamma) \cap \{z\} = \emptyset \tag{122}$$
$$\rho.f = \rho' \tag{123}$$
$$\tag{124}$$

We split the composition in 120 as

$$(\sigma_1, O_1, U_1, T_1, F_1) \in [\![\Gamma, z : \mathsf{rcultr}\ \_, x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} \tag{125}$$
$$(\sigma_2, O_2, U_2, T_2, F_2) = m \tag{126}$$
$$\sigma_1 \bullet \sigma_2 = \sigma \tag{127}$$
$$O_1 \bullet_O O_2 = O \tag{128}$$
$$U_1 \cup U_2 = U \tag{129}$$
$$T_1 \cup T_2 = T \tag{130}$$
$$F_1 \uplus F_2 = F \tag{131}$$
$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1) \tag{132}$$
$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2) \tag{133}$$

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

$$(\sigma', O_1', U_1', T_1', F_1) \in [\![\Gamma, x : \mathsf{rcultr}\, \rho \mathcal{N}[f \mapsto z], z : \mathsf{rcultr}\, \rho' \mathcal{N}_\emptyset]\!]_{M,tid} \tag{134}$$
$$\mathcal{N}(f) = z \tag{135}$$
$$(\sigma', O_2', U_2', T_2', F_2) \in \mathcal{R}(\{m\}) \tag{136}$$
$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{137}$$
$$O_1' \bullet_O O_2' = O' \tag{138}$$
$$U_1' \cup U_2' = U' \tag{139}$$
$$T_1' \cup T_2' = T' \tag{140}$$
$$F_1' \uplus F_2' = F' \tag{141}$$
$$\mathsf{WellFormed}(\sigma_1', O_1', U_1', T_1') \tag{142}$$
$$\mathsf{WellFormed}(\sigma_2', O_2', U_2', T_2') \tag{143}$$

Let $h(s(z, tid), f)$ be o_z. We also know from operational semantics that the machine state has changed as

$$\sigma_1' = \sigma_1[s(x, tid) \mapsto o_z] \tag{144}$$

135 is determined directly from operational semantics.

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

$$O_1' = O_1 \tag{145}$$

146 follows from 120

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{146}$$

σ_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 125 and choices on maps makes $(\sigma_1', O_1', U_1', T_1')$ in denotation

$$[\![\Gamma, x : \mathsf{rcultr}\,\rho\,\mathcal{N}[f \mapsto z], z : \mathsf{rcultr}\,\rho'\,\mathcal{N}_\emptyset]\!]_{M,tid}$$

In the rest of the proof, we prove 142, 143 and show the composition of $(\sigma_1', O_1', U_1', T_1')$ and $(\sigma_2', O_2', U_2', T_2')$.

To prove 136, we need to show that each of the memory axioms in Section A.2 holds for the state $(\sigma', O_1', U_1', T_1')$.

Case 31. - **UNQR** By 144, 132.**UNQR** and $\sigma.rt = \sigma.rt'$.

Case 32. - **OW** By 144, 145 and 132.**OW**

Case 33. - **RWOW** By 144, 145 and 132.**RWOW**

Case 34. - **AWRT** Trivial.

Case 35. - **IFL** By 134, 132.**WULK**, 145, choice of F_1' and operational semantics.

Case 36. - **FLR** By operational semantics(144), choice for F_1' and 132.

Case 37. - **WULK** By 132.**WULK**, 145 and operational semantics($\sigma.l = \sigma.l'$).

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

Case 39. - **HD**

Case 40. - **WNR** By 146 and operational semantics($\sigma.l = \sigma.l'$).

Case 41. - **RINFL** By operational semantics(144) bounding threads have not changed. We choose F_1' as F_1. These two together with 132 shows **RINFL**.

Case 42. - **ULKR** Trivial.

Case 43. - **UNQRT** By 132.**UNQRT**, 145 and 144.

To prove 136, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{147}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{148}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{149}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{150}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R_2 = \sigma_2'.R_2 \wedge \sigma_2.rt = \sigma_2'.rt \tag{151}$$

$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{152}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h) \tag{153}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h') \tag{154}$$

To prove all relations (147-152) we assume 146 which is to assume T_2 as subset of reader threads. Let σ_2' be σ_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 133 we show 143. 149 and 150 follows trivially. 152 follows from choice of σ_2' and 144(determined by operational semantics).

147 and 148 follow from 146 and choice of F_2'.151, 153 and 154 are determined by choice of σ_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 127. 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 (WriteFreshField).

$$[\![p.f := z]\!](\lfloor [\![\Gamma, p : \mathit{rcuFresh}\,\mathcal{N}'_{f,\emptyset},\ x : \mathit{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} * \{m\}\rfloor) \subseteq$$
$$\lfloor [\![\Gamma\,, p : \mathit{rcuFresh}\ \ \mathcal{N}(\cup_{f \rightharpoonup z})\,, x : \mathit{rcultr}\,\rho\,\mathcal{N}([f \rightharpoonup z])]\!] * \mathcal{R}(\{m\})\rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F) \in [\![\Gamma, p : \mathsf{rcuFresh}\,\mathcal{N}'_{f,\emptyset},\ x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} * \{m\} \tag{155}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{156}$$

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

$$z : \mathsf{rcultr}\,\rho.f\ \_ \text{ and } \mathcal{N}(f) = z \text{ and } f \notin dom(\mathcal{N}') \tag{157}$$

We split the composition in 155 as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![\Gamma, p : \mathsf{rcuFresh}\,\mathcal{N}'_{f,\emptyset},\ x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} \tag{158}$$
$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{159}$$
$$\sigma_1 \bullet \sigma_2 = \sigma \tag{160}$$
$$O_1 \bullet_O O_2 = O \tag{161}$$
$$U_1 \cup U_2 = U \tag{162}$$
$$T_1 \cup T_2 = T \tag{163}$$
$$F_1 \uplus F_2 = F \tag{164}$$
$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1) \tag{165}$$
$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{166}$$

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

$$(\sigma', O_1', U_1', T_1', F_1') \in [\![\Gamma, p : \mathsf{rcuFresh}\ \mathcal{N}(\cup_{f \rightharpoonup z}), x : \mathsf{rcultr}\,\rho\mathcal{N}([f \rightharpoonup z])]\!]_{M,tid} \tag{167}$$

$$\mathcal{N}(f) = z \wedge \mathcal{N}'(f) = z \tag{168}$$

$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{169}$$

$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{170}$$

$$O_1' \bullet_O O_2' = O' \tag{171}$$

$$U_1' \cup U_2' = U' \tag{172}$$

$$T_1' \cup T_2' = T' \tag{173}$$

$$F_1' \uplus F_2' = F' \tag{174}$$

$$\mathsf{WellFormed}(\sigma_1', O_1', U_1', T_1', F_1') \tag{175}$$

$$\mathsf{WellFormed}(\sigma_2', O_2', U_2', T_2', F_2') \tag{176}$$

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

$$\sigma' = \sigma[h(s(p, tid), f) \mapsto s(z, tid)] \tag{177}$$

There exists no change in the observation of heap locations

$$O_1' = O_1 \tag{178}$$

179 follows from 155

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{179}$$

σ_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 158 and choices on maps makes $(\sigma_1', O_1', U_1', T_1')$ in denotation

$$[\![\Gamma, p : \mathsf{rcuFresh}\ \mathcal{N}(\cup_{f \rightharpoonup z}), x : \mathsf{rcultr}\,\rho\mathcal{N}([f \rightharpoonup z])]\!]_{M,tid}$$

In the rest of the proof, we prove 175 and 176 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 175, 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 165.**UNQR**, 175.**FR**(proved) and $\sigma.rt = \sigma.rt'$.

Case 45. - **OW** By 177,178 and 165.**OW**

Case 46. - **RWOW** By 177, 178 and 165.**RWOW**

Case 47. - **AWRT** Trivial.

Case 48. - **IFL** By 165.**WULK**, 178, choice of F_1' and operational semantics.

Case 49. - **FLR** By operational semantics(177), choice of F_1' and 165.

Case 50. - **WULK** By 165.**WULK**, 178 and operational semantics($\sigma.l = \sigma.l'$).

Case 51. - **WF** By 165.**WF**, 179, 178 and operational semantics(177).

Case 52. - **FR** By 165.**FR**, 179, 178 and operational semantics(177).

Case 53. - **FNR** By 165.**FNR**, 179, 178 and operational semantics(177).

Case 54. - **FPI** By 165.**FPI**, 158 and 157

Case 55. - **HD**

Case 56. - **WNR** By 179 and operational semantics($\sigma.l = \sigma.l'$).

Case 57. - **RINFL** By operational semantics(177 - bounding threads have not changed), choice of F_1' and 165.

Case 58. - **ULKR** Trivial.

Case 59. - **UNQRT** By 165.**UNQRT**, 178 and 177.

To prove 169, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{180}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{181}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{182}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{183}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{184}$$

$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{185}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h) \tag{186}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h') \tag{187}$$

To prove all relations (180-185) we assume 179 which is to assume T_2 as subset of reader threads and 166. Let σ_2' be σ_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 166 we show 176. 182 and 183 follows trivially. 185 follows from choice of σ_2' and 177(determined by operational semantics).

180 and 181 follow from 179 and choice of F_2'. 184 are determined by operational semantics, choice of σ_2' and choices made on maps related to the assertions.

172-174 are trivial by choices on related maps and semantics of the composition operators for these maps. 186 and 187 follow from choice of σ_2'.

$O_1' \bullet O_2'$ follows from 161, 178 and choice of O_2.

We assume $\sigma_1.h \bullet \sigma_2.h$. We know from 157 that $f \notin dom(\mathcal{N}')$. From 167, 175-176.**FNR**, 175-176.**RITR** and 175-176.**WNR** we show $\sigma_1'.h \bullet \sigma_2'.h$ (with choices for other maps in the machine state we show 137). All compositions shown let us to derive conclusion for $(\sigma_1', O_1', U_1', T_1') \bullet (\sigma_2', O_2', U_2', T_2')$.

Lemma 20 (Sycn).

$$[\![\mathit{SyncStart}; \mathit{SyncStop}]\!](\lfloor [\![\Gamma]\!]_{M,tid} * \{m\}\rfloor) \subseteq \lfloor [\![\Gamma[\overline{x : \mathit{freeable}/x : \mathit{unlinked}}]]\!] * \mathcal{R}(\{m\})\rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F) \in [\![\Gamma]\!]_{M,tid} * \{m\} \tag{188}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{189}$$

We split the composition in 188 as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![\Gamma]\!]_{M,tid} \tag{190}$$
$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{191}$$
$$\sigma_1 \bullet \sigma_2 = \sigma \tag{192}$$
$$O_1 \bullet_O O_2 = O \tag{193}$$
$$U_1 \cup U_2 = U \tag{194}$$
$$T_1 \cup T_2 = T \tag{195}$$
$$F_1 \uplus F_2 = F \tag{196}$$
$$\mathsf{WellFormed}(\sigma, O_1, U_1, T_1, F_1) \tag{197}$$
$$\mathsf{WellFormed}(\sigma, O_2, U_2, T_2, F_2) \tag{198}$$

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

$$(\sigma', O'_1, U'_1, T'_1, F'_1) \in [\![\Gamma[\overline{x : \mathsf{freeable}/x : \mathsf{unlinked}}]]\!]_{M,tid} \tag{199}$$
$$(\sigma', O'_2, U'_2, T'_2, F'_2) \in \mathcal{R}(\{m\}) \tag{200}$$
$$\sigma'_1 \bullet \sigma'_2 = \sigma' \tag{201}$$
$$O'_1 \bullet_O O'_2 = O' \tag{202}$$
$$U'_1 \cup U'_2 = U' \tag{203}$$
$$T'_1 \cup T'_2 = T' \tag{204}$$
$$\tag{205}$$
$$F'_1 \uplus F'_2 = F' \tag{206}$$
$$\mathsf{WellFormed}(\sigma', O'_1, U'_1, T'_1, F'_1) \tag{207}$$
$$\mathsf{WellFormed}(\sigma', O'_2, U'_2, T'_2, F'_2) \tag{208}$$

We also know from operational semantics that `SyncStart` changes

$$\sigma'_1.B = \sigma_1.B[\emptyset \mapsto R] \tag{209}$$

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

$$\sigma'_1 = \sigma_1 \tag{210}$$

There exists no change in the observation of heap locations

$$O_1' = O_1(\forall_{x \in \Gamma}.\, s(x, tid))[\mathsf{unlinked} \mapsto \mathsf{freeable}] \tag{211}$$

and we pick free list to be

$$F_1' = F_1(\forall_{x:\mathsf{unlinked} \in \Gamma, T \subseteq R}.\, s(x, tid)[T \mapsto \{\emptyset\}]) \tag{212}$$

213 follows from 188

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{213}$$

Let T_1' be T_1 and σ_1'(not changed) be determined by operational semantics. The undefined map need not change so we can pick U_1' as U_1. Assuming 190 and choices on maps makes $(\sigma_1', O_1', U_1', T_1', F_1')$ in denotation

$$[\![\Gamma[\overline{x : \mathsf{freeable}/x : \mathsf{unlinked}}]]\!]_{M,tid}$$

In the rest of the proof, we prove 207 and 208 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 207, 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 197. We also know 199(as we showed the support of state to the denotation).

To prove 200, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{214}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{215}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{216}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{217}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{218}$$

$$\forall x, t \in T_2.\, \sigma_2.s(x, t) = \sigma_2'.s(x, t) \tag{219}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h) \tag{220}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h') \tag{221}$$

To prove all relations (214-219) we assume 213 which is to assume T_2 as subset of reader threads and 198. Let σ_2' be σ_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 198 we show 208. 216 and 217 follows trivially. 219 follows from choice of σ_2' and 210(determined by operational semantics).

214 and 215 follow from 213. 218 are determined by choice of σ_2' and operational semantics and choices made on maps related to the assertions.

203-206 follow from 193-196 trivially by choices on maps of logical state and semantics of composition operators. 201 follow from 192, 210-213 and choice of σ_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 (Alloc).

$$[\![x := new]\!](\lfloor [\![\Gamma, x : \mathit{undef}]\!]_{M,tid} * \{m\}\rfloor) \subseteq \lfloor [\![\Gamma, x : \mathit{rcuFresh}\,\mathcal{N}_\emptyset]\!] * \mathcal{R}(\{m\})\rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F) \in [\![\Gamma, x : \mathsf{undef}]\!]_{M,tid} * \{m\}\rfloor) \tag{222}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{223}$$

We split the composition in 222 as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![\Gamma, x : \mathsf{undef}]\!]_{M,tid} \tag{224}$$
$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{225}$$
$$\sigma_1 \bullet \sigma_2 = \sigma \tag{226}$$
$$O_1 \bullet_O O_2 = O \tag{227}$$
$$U_1 \cup U_2 = U \tag{228}$$
$$T_1 \cup T_2 = T \tag{229}$$
$$F_1 \uplus F_2 = F \tag{230}$$
$$\mathsf{WellFormed}(\sigma, O_1, U_1, T_1, F_1) \tag{231}$$
$$\mathsf{WellFormed}(\sigma, O_2, U_2, T_2, F_2) \tag{232}$$

We must show $\exists_{O_1', O_2', U_1', U_2', T_1', T_2', F_1', F_2'}$ such that

$$(\sigma', O_1', U_1', T_1', F_1') \in [\![\Gamma, x : \mathsf{rcuFresh}\,\mathcal{N}_\emptyset]\!] \tag{233}$$
$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{234}$$
$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{235}$$
$$O_1' \bullet_O O_2' = O' \tag{236}$$
$$U_1' \cup U_2' = U' \tag{237}$$
$$T_1' \cup T_2' = T' \tag{238}$$
$$F_1' \uplus F_2' = F' \tag{239}$$
$$\mathsf{WellFormed}(\sigma', O_1', U_1', T_1') \tag{240}$$
$$\mathsf{WellFormed}(\sigma', O_2', U_2', T_2') \tag{241}$$

From operational semantics we know that $s(y, tid)$ is ℓ. We also know from operational semantics that the machine state has changed as

$$\sigma' = \sigma[h(\ell) \mapsto \mathsf{nullmap}] \tag{242}$$

There exists no change in the observation of heap locations

$$O_1' = O_1(\ell)[\mathsf{undef} \mapsto \mathsf{fresh}] \tag{243}$$

244 follows from 222

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{244}$$

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(233) that $(\sigma', O_1', U_1', T_1', F_1')$ is in denotation of

$$[\![\Gamma, x : \mathsf{rcuFresh}\, \mathcal{N}_\emptyset]\!]$$

In the rest of the proof, we prove 240, 241 and $(\sigma_1', O_1', U_1', T_1', F_1')$ and $(\sigma_2', O_2', U_2', T_2', F_2')$. To prove 240, 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 60. - **UNQR** Determined by 233 and operational semantics(ℓ is fresh-unique).

Case 61. - **RWOW**, **OW** By 233

Case 62. - **AWRT** Determined by operational semantics(ℓ is fresh-unique).

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

Case 64. - **FLR** determined by operational semantics and 233.

Case 65. - **WF** By 244, 233 and 243.

Case 66. - **FR** Determined by operational semantics(ℓ is fresh-unique).

Case 67. - **FNR** By 233 and operational semantics(ℓ is fresh-unique).

Case 68. - **FPI** By 233 and $\mathcal{N}_{f,\emptyset}$.

Case 69. - **HD**

Case 70. - **WNR** By 244.

To prove 234, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{245}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{246}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{247}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{248}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{249}$$

$$\forall x, t \in T.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{250}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h) \tag{251}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h') \tag{252}$$

To prove all relations (245-252) we assume 244 which is to assume T_2 as subset of reader threads and 232. Let σ_2' be σ_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 232 and choices on maps we show 241. 247 and 248 follow trivially. 250 follows from choice of σ_2' and 242(determined by operational semantics). 236-239 follow from 227-230, semantics of compositions operators and choices made for maps of the logical state.

245 and 246 follow from 244 and choice on F_2'. 249 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(ℓ is unique and fresh). So, 235 follows from 226 and choice of σ_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 22 (Free).

$$[\![Free(x)]\!](\lfloor [\![x : \mathit{freeable}]\!]_{M,tid} * \{m\}\rfloor) \subseteq \lfloor [\![x : \mathit{undef}]\!] * \mathcal{R}(\{m\})\rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F) \in [\![x : \mathsf{freeable}]\!]_{M,tid} * \{m\}\rfloor) \tag{253}$$

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{254}$$

We split the composition in 253 as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![x : \mathsf{freeable}]\!]_{M,tid} \tag{255}$$

$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{256}$$

$$\sigma_1 \bullet \sigma_2 = \sigma \tag{257}$$

$$O_1 \bullet_O O_2 = O \tag{258}$$

$$U_1 \cup U_2 = U \tag{259}$$

$$T_1 \cup T_2 = T \tag{260}$$

$$F_1 \uplus F_2 = F \tag{261}$$

$$\mathsf{WellFormed}(\sigma, O_1, U_1, T_1, F_1) \tag{262}$$

$$\mathsf{WellFormed}(\sigma, O_2, U_2, T_2, F_2) \tag{263}$$

We must show $\exists_{O_1',O_2',U_1',U_2',T_1',T_2',F_1',F_2'}$ such that

$$(\sigma', O_1', U_1', T_1', F_1') \in [\![x : \mathsf{undef}]\!] \tag{264}$$
$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{265}$$
$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{266}$$
$$O_1' \bullet_O O_2' = O' \tag{267}$$
$$U_1' \cup U_2' = U' \tag{268}$$
$$T_1' \cup T_2' = T' \tag{269}$$
$$F_1' \uplus F_2' = F' \tag{270}$$
$$\mathsf{WellFormed}(\sigma', O_1', U_1', T_1') \tag{271}$$
$$\mathsf{WellFormed}(\sigma', O_2', U_2', T_2') \tag{272}$$

From operational semantics we know that

$$\sigma_1' = \sigma_1 \tag{273}$$

There exists no change in the observation of heap locations

$$O_1' = O_1(s(x, tid))[\mathsf{freeable} \mapsto \mathsf{undef}] \tag{274}$$

$$F_1' = F_1 \setminus \{s(x, tid) \mapsto \{\emptyset\}\} \tag{275}$$

$$U_1' = U_1 \cup \{(x, tid)\} \tag{276}$$

277 follows from 253

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{277}$$

Let T_1' to be T_1. All 273-276 show(264) that $(\sigma', O_1', U_1', T_1', F_1')$ is in denotation of

$$[\![x : \mathsf{undef}]\!]$$

In the rest of the proof, we prove 271, 27224, 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 271, 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 it trivial by 273-276 and 264.

To prove 265, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{278}$$
$$l \in T_2 \rightarrow F_2 = F_2' \tag{279}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{280}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{281}$$
$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{282}$$
$$\forall x, t \in T.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{283}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h) \tag{284}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h') \tag{285}$$

To prove all relations (278-285) we assume 277 which is to assume T_2 as subset of reader threads and 232. Let σ_2' be σ_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 232 and choices on maps we show 272. 280 and 281 follow trivially. 283 follows from choice of σ_2' and 273(determined by operational semantics). 267-270 follow from 259-261, semantics of composition operators and choices on related maps.

278 and 279 follow from 277 and choice on F_2'. 282 are determined by operational semantics, choice of σ_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 undef) and this makes showing 267 straightforward. Since other machine components do not change(determined by operational semantics), 266 follows from 257, 273 and choice of σ_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 23 (RReadStack).

$$[\![z := x]\!](\lfloor [\![\Gamma, z : \mathit{rcultr}, x : \mathit{rcultr}]\!]_{R,tid} * \{m\} \rfloor) \subseteq \lfloor [\![\Gamma, x : \mathit{rcultr}, z : \mathit{rcultr}]\!] * \mathcal{R}(\{m\}) \rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F) \in [\![\Gamma, \Gamma, z : \mathsf{rcultr}, x : \mathsf{rcultr}]\!]_{R,tid} * \{m\} \tag{286}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{287}$$

We split the composition in 286 as

$$(\sigma_1, O_1, U_1, T_1, F_1) \in [\![\Gamma, \Gamma, z : \mathsf{rcultr}, x : \mathsf{rcultr}]\!]_{R,tid} \tag{288}$$
$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{289}$$
$$O_1 \bullet_O O_2 = O \tag{290}$$
$$\sigma_1 \bullet \sigma_2 = \sigma \tag{291}$$
$$U_1 \cup U_2 = U \tag{292}$$
$$T_1 \cup T_2 = T \tag{293}$$
$$F_1 \uplus F_2 = F \tag{294}$$
$$\mathsf{WellFormed}(\sigma, O_1, U_1, T_1, F_1) \tag{295}$$
$$\mathsf{WellFormed}(\sigma, O_2, U_2, T_2, F_2) \tag{296}$$

We must show $\exists_{O_1', O_2', U_1', U_2', T_1', T_2', F_1', F_2'}$ such that

$$(\sigma', O_1', U_1', T_1', F_1') \in [\![\Gamma, x : \mathsf{rcultr}, z : \mathsf{rcultr}]\!]_{R,tid} \tag{297}$$
$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{298}$$
$$O_1' \bullet_O O_2' = O' \tag{299}$$
$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{300}$$
$$U_1' \cup U_2' = U' \tag{301}$$
$$T_1' \cup T_2' = T' \tag{302}$$
$$F_1' \uplus F_2' = F' \tag{303}$$
$$\mathsf{WellFormed}(\sigma', O_1', U_1', T_1', F_1') \tag{304}$$
$$\mathsf{WellFormed}(\sigma', O_2', U_2', T_2', F_2') \tag{305}$$

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

$$\sigma_1' = \sigma_1 \tag{306}$$

There exists no change in the observation of heap locations

$$O_1' = O_1 \tag{307}$$

308 follows from 286

$$T_1 \subseteq R \tag{308}$$

Let T_1' be T_1 and σ_1' be determined by operational semantics as σ_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 288 and choices on maps makes $(\sigma_1', O_1', U_1', T_1', F_1')$ in denotation

$$[\![\Gamma, x : \mathsf{rcultr}, z : \mathsf{rcultr}]\!]_{R,tid}$$

In the rest of the proof, we prove 304, 30524, 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 304, 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 295 and knowing 308, 307 and components of the state determined by operational semantics.

To prove 305, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{309}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{310}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{311}$$

$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{312}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.B = \sigma_2'.B \wedge \sigma_2.rt = \sigma_2'.rt \tag{313}$$

$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{314}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h) \tag{315}$$

$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h') \tag{316}$$

σ_2, O_2, U_2 and T_2 need not change so that we choose σ_2' to be σ_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 309-316 trivial and 299-302 follow from assumptions 290-294, 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 24 (RReadHeap).

$$[\![z := x.f]\!](\lfloor [\![\Gamma, z : \mathit{rcultr}, x : \mathit{rcultr}]\!]_{R,tid} * \{m\}\rfloor) \subseteq \lfloor [\![\Gamma, x : \mathit{rcultr}, z : \mathit{rcultr}]\!] * \mathcal{R}(\{m\})\rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F) \in [\![\Gamma, z : \mathsf{rcultr}, x : \mathsf{rcultr}]\!]_{R,tid} * \{m\}\rfloor) \tag{317}$$

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{318}$$

We split the composition in 317 as

$$(\sigma_1, O_1, U_1, T_1, F_1) \in [\![\Gamma, z : \mathsf{rcultr}, x : \mathsf{rcultr}]\!]_{R,tid} \tag{319}$$

$$(\sigma_2, O_2, U_2, T_2, F_2) = m \tag{320}$$

$$\sigma_1 \bullet \sigma_2 = \sigma \tag{321}$$

$$O_1 \bullet_O O_2 = O \tag{322}$$

$$U_1 \cup U_2 = U \tag{323}$$

$$T_1 \cup T_2 = T \tag{324}$$

$$F_1 \uplus F_2 = F \tag{325}$$

$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1) \tag{326}$$

$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{327}$$

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

$$(\sigma_1', O_1', U_1', T_1', F_1') \in \lfloor [\![\Gamma, x : \mathsf{rcultr}, z : \mathsf{rcultr}]\!] \tag{328}$$
$$(\sigma_2', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{329}$$
$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{330}$$
$$O_1' \bullet_O O_2' = O' \tag{331}$$
$$U_1' \cup U_2' = U' \tag{332}$$
$$T_1' \cup T_2' = T' \tag{333}$$
$$\mathsf{WellFormed}(\sigma_1', O_1', U_1', T_1', F_1') \tag{334}$$
$$\mathsf{WellFormed}(\sigma_2', O_2', U_2', T_2', F_2') \tag{335}$$

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

$$\sigma_1' = \sigma_1[s(z, tid) \mapsto o_x] \tag{336}$$

There exists no change in the observation of heap locations

$$O_1' = O_1 \tag{337}$$

338 follows from 317

$$T_1 \subseteq R \tag{338}$$

Proof is similar to Lemma 23.

A.4 Soundness Proof of Structural Program Actions

In this section, we introduce soundness Theorem 3 for structural rules of the type system. We consider the cases of the induction on derivation of $\Gamma \vdash C \dashv \Gamma$ 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 rools.

Theorem 3 (Type System Soundness).

$$\forall_{\Gamma,\Gamma',C}. \Gamma \vdash C \dashv \Gamma' \implies [\![\Gamma \vdash C \dashv \Gamma']\!]$$

Proof. Induction on derivation of $\Gamma \vdash_M C \dashv \Gamma$.

Case 71. -**M**: consequence where C has the form $\Gamma \vdash_M C \dashv \Gamma'''$. We know

$$\Gamma' \vdash_M C \dashv \Gamma'' \tag{339}$$
$$\Gamma \prec: \Gamma' \tag{340}$$
$$\Gamma'' \prec: \Gamma''' \tag{341}$$
$$\{[\![\Gamma']\!]_{M,tid}\} C \{[\![\Gamma'']\!]_{M,tid}\} \tag{342}$$

We need to show

$$\{[\![\Gamma]\!]_{M,tid}\}C\{[\![\Gamma'''']\!]_{M,tid}\} \tag{343}$$

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

From 340 and Lemma 25 we know

$$[\![\Gamma]\!]_{M,tid} \sqsubseteq [\![\Gamma']\!]_{M,tid} \tag{344}$$

From 341 and 25 we know

$$[\![\Gamma'']\!]_{M,tid} \sqsubseteq [\![\Gamma''']\!]_{M,tid} \tag{345}$$

By using 344, 345 and 342 as antecedentes of Views Logic's consequence rule, we conclude 343.

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

$$\{[\![\Gamma]\!]_{M,tid}\} \vdash_M C_1; C_2 \dashv \{[\![\Gamma'']\!]_{M,tid}\} \tag{346}$$

We know

$$\Gamma \vdash_M C_1 \dashv \Gamma' \tag{347}$$
$$\Gamma' \vdash_M C_2 \dashv \Gamma'' \tag{348}$$
$$\{[\![\Gamma]\!]_{M,tid}\}C_1\{[\![\Gamma']\!]_{M,tid}\} \tag{349}$$
$$\{[\![\Gamma']\!]_{M,tid}\}C_2\{[\![\Gamma'']\!]_{M,tid}\} \tag{350}$$

By using 349 and 350 as the antecedents for the Views sequencing rule, we can derive the conclusion for 346.

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

$$\Gamma \vdash_M C \dashv \Gamma \tag{351}$$
$$\Gamma(x) = \mathsf{bool} \tag{352}$$
$$\{[\![\Gamma]\!]_{M,tid}\}C\{[\![\Gamma]\!]_{M,tid}\} \tag{353}$$

Our goal is to prove

$$\{[\![\Gamma]\!]_{M,tid}\}\,(assume\,(x)\,;C)^*\,;assume(\neg x)\{[\![\Gamma]\!]_{M,tid}\} \tag{354}$$

We prove 354 by from the consequence rule, based on the proofs of the following 355 and 356

$$\{[\![\Gamma]\!]_{M,tid}\}\,(assume\,(x)\,;C)^*\,\{[\![\Gamma]\!]_{M,tid}\} \tag{355}$$

$$\{[\![\Gamma]\!]_{M,tid}\} assume\,(\neg x)\,\{[\![\Gamma]\!]_{M,tid}\} \tag{356}$$

The poof of 355 follows from Views Logic's proof rule for assume construct by using

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

as antecedant. We can use this antecedant together with the antecedent we know from 353

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

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

$$\{[\![\Gamma]\!]_{M,tid}\} assume\,(x)\,;C\{[\![\Gamma]\!]_{M,tid}\}$$

to the proof rule for nondeterministic looping.

The proof of 356 follows from Views Logic's proof rule for assume construct by using the

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

as the antecedent.

Case 74. -**M**: where C is a loop statement. C has the form $while(x.f \neq \texttt{null})\{C\}$ Proof is similar to the one for T-Loop1.

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

$$\Gamma, x : \mathsf{rcultr}\ \rho\ \mathcal{N}([f_1 \rightharpoonup z]) \vdash_M C_1 \dashv \Gamma' \tag{357}$$

$$\Gamma, x : \mathsf{rcultr}\ \rho\ \mathcal{N}([f_2 \rightharpoonup z]) \vdash_M C_2 \dashv \Gamma' \tag{358}$$

$$\{[\![\Gamma,\ x : \mathsf{rcultr}\ \rho\ \mathcal{N}([f_1 \rightharpoonup z])]\!]_{M,tid}\} C_1 \{[\![\Gamma']\!]_{M,tid}\} \tag{359}$$

$$\{[\![\Gamma,\ x : \mathsf{rcultr}\ \rho\ \mathcal{N}([f_2 \rightharpoonup z])]\!]_{M,tid}\} C_2 \{[\![\Gamma']\!]_{M,tid}\} \tag{360}$$

Our goal is to prove

$$\begin{array}{l}\{[\![\Gamma, x : \mathsf{rcultr}\ \rho\ \mathcal{N}([f_1 | f_2 \rightharpoonup z])]\!]_{M,tid}\} \\ y = x.f_1; (assume\,(z = y)\,; C_1) + (assume\,(y \neq z)\,; C_2) \\ \{[\![\Gamma']\!]_{M,tid}\}\end{array} \tag{361}$$

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 prove361 from the consequence rule of Views Logic based on the proofs of the following 362 and 363

$$\begin{array}{l}\{[\![\Gamma, x : \mathsf{rcultr}\ \rho\ \mathcal{N}([f_1 | f_2 \rightharpoonup z])]\!]_{M,tid}\} \\ (assume\,(z = y)\,; C_1) + (assume\,(y \neq z)\,; C_2) \\ \{[\![\Gamma']\!]_{M,tid}\}\end{array} \tag{362}$$

and

$$\begin{array}{l}\{[\![\Gamma, x : \mathsf{rcultr}\ \rho\ \mathcal{N}[f_1 | f_2 \rightharpoonup z]]\!]_{M,tid}\} \\ y = x.f_1 \\ \{[\![\Gamma, x : \mathsf{rcultr}\ \rho\ \mathcal{N}([f_1 | f_2 \rightharpoonup z])[\![_{M,tid} \cap [\![x : \mathsf{rcultr}\ \rho\ \mathcal{N}([f_1 \rightharpoonup y])]\!]_{M,tid}\}\end{array} \tag{363}$$

363 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 362 from the branch rule of Views Logic based on the proofs of the following 364 and 365

$$\begin{array}{l}\{[\![\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1|f_2 \rightharpoonup z])]\!]_{M,tid} \cap \\ [\![x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 \rightharpoonup y])]\!]_{M,tid}\} \\ (assume\,(z = y)\,; C_1) \\ \{[\![\Gamma']\!]_{M,tid}\}\end{array} \tag{364}$$

and

$$\begin{array}{l}\{[\![\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1|f_2 \rightharpoonup z])]\!]_{M,tid} \cap \\ [\![x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 \rightharpoonup y])]\!]_{M,tid}\} \\ (assume\,(z \neq y)\,; C_2) \\ [\![\Gamma']\!]_{M,tid}\}\end{array} \tag{365}$$

We show 364 from Views Logic's proof rule for the assume construct by using

$$\begin{array}{l}\{[\![\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1|f_2 \rightharpoonup z])]\!]_{M,tid} \cap \\ [\![x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 \rightharpoonup y])]\!]_{M,tid}\} \\ assume\,(y = z) \\ \{[\![\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 \rightharpoonup z])]\!]_{M,tid}\}\end{array}$$

as the antecedent. We can use this antecedent together with

$$\{[\![\Gamma, x : \mathsf{rcultr} \rho \mathcal{N}([f_1 \rightharpoonup z])]\!]_{M,tid}\} C_1 \{[\![\Gamma']\!]_{M,tid}\}$$

as antecedents to the View's Logic's proof rule for sequencing.

We show 365 from Views Logic's proof rule for the assume construct by using

$$\begin{array}{l}\{[\![\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1|f_2 \rightharpoonup z]) \cap \\ x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 \rightharpoonup y])]\!]_{M,tid}\} \\ assume(x \neq y) \\ \{[\![\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_2 \rightharpoonup z])]\!]_{M,tid}\}\end{array}$$

as the antecedent. We can use this antecedent together with

$$\{[\![\Gamma, x : \mathsf{rcultr} \rho \mathcal{N}([f_2 \rightharpoonup z])]\!]_{M,tid}\} C_2 \{[\![\Gamma']\!]_{M,tid}\}$$

as antecedents to the Views Logic's proof rule for sequencing.

Case 76. -**M**: case where C is branch statement. C has the form $if(x.f == \texttt{null})then\{C_1\}else\{C_2\}$. Proof is similar to one for T-Branch1.

Case 77. -**O**: parallel where C has the form $\Gamma_1, \Gamma_2 \vdash_O C_1||C_2 \dashv \Gamma_1', \Gamma_2'$ We know

$$\Gamma_1 \vdash C_1 \dashv \Gamma_1' \tag{366}$$

$$\Gamma_2 \vdash C_2 \dashv \Gamma_2' \tag{367}$$

$$\{[\![\Gamma_1]\!]\} C_1 \{[\![\Gamma_1']\!]\} \tag{368}$$

$$\{[\![\Gamma_2]\!]\} C_2 \{[\![\Gamma_2']\!]\} \tag{369}$$

We need to show

$$\{[\![\Gamma_1, \Gamma_2]\!]\}C_1||C_2\{[\![\Gamma_1', \Gamma_2']\!]\} \tag{370}$$

By using 368 and 369 as antecedents to Views Logic's parallel rule, we can draw conclusion for 371

$$\{[\![\Gamma_1]\!] * [\![\Gamma_2]\!]\}C_1||C_2\{[\![\Gamma_1']\!] * [\![\Gamma_2']\!]\} \tag{371}$$

Showing 370 requires showing

$$[\![\Gamma_1, \Gamma_2]\!] \sqsubseteq [\![\Gamma_1]\!] * [\![\Gamma_2]\!] \tag{372}$$

$$[\![\Gamma_1']\!] * [\![\Gamma_2']\!] \sqsubseteq [\![\Gamma_1', \Gamma_2']\!] \tag{373}$$

By using 372 and 373(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 370.

Case 78. -**M** where C has form $\mathsf{RCUWrite}\, x.f$ as y in $\bar{s}$ which desugars into

$$\mathsf{WriteBegin}; x.f := y; \bar{s}; \mathsf{WriteEnd}$$

We assume from the rule ToRCUWrite

$$\Gamma, y : \mathsf{rcultr}\ \_ \vdash_M \bar{s} \dashv \Gamma' \tag{374}$$
$$\mathsf{FType}(f) = \mathsf{RCU} \tag{375}$$
$$\mathsf{NoFresh}(\Gamma') \tag{376}$$
$$\mathsf{NoUnlinked}(\Gamma') \tag{377}$$
$$\{[\![\Gamma, y : \mathsf{rcultr}\ \_]\!]_{M,tid}\}\bar{s}\{[\![\Gamma']\!]_{M,tid}\} \tag{378}$$

Our goal is to prove

$$\{[\![\Gamma]\!]_{M,tid}\}\mathsf{WriteBegin}; x.f := y; \mathsf{WriteEnd}\{[\![\Gamma']\!]_{M,tid}\} \tag{379}$$

Proof starts with application of the sequence rule. Assumptions 376-377 guarantee that there exists no change in the state(no heap update) due to any action in the body $\bar{s}$ which includes $x.f := y$. 379 follows from assumptions 374-378 trivially.

Lemma 25 (Context-SubTyping-M).

$$\Gamma \prec: \Gamma' \implies [\![\Gamma]\!]_{M,tid} \sqsubseteq [\![\Gamma']\!]_{M,tid}$$

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

Lemma 26 (Context-SubTyping-R).

$$\Gamma \prec: \Gamma' \implies [\![\Gamma]\!]_{R,tid} \sqsubseteq [\![\Gamma']\!]_{R,tid}$$

Proof. Induction on the subtyping derivation. Then inducting on the first entry in the non-empty context(empty case is trivial) which follows from 28.

Lemma 27 (Singleton-SubTyping-M).

$$x : T \prec: x : T' \implies [\![x : T]\!]_{M,tid} \sqsubseteq [\![x : T']\!]_{R,tid}$$

Proof. Proof by case analysis on structure of T' and T. Important case includes the subtyping relation is defined over components of rcultr type. T' including approximation on the path component

$$\rho.f_1 \prec: \rho.f_1|f_2$$

together with the approximation on the field map

$$\mathcal{N}([f_1 \rightharpoonup \_]) \prec: \mathcal{N}([f_1|f_2 \rightharpoonup \_])$$

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 28 (Singleton-SubTyping-R).

$$x : T \prec: x : T' \implies [\![x : T]\!]_{M,tid} \sqsubseteq [\![x : T']\!]_{M,tid}$$

Proof. Proof is similar to 27 with a single trivial reflexive derivation relation (T-TSub2)

$$\mathsf{rcultr} \prec: \mathsf{rcultr}$$

B RCU BST Delete

$void\ delete(\ \texttt{int}\ data)\ \{$
$WriteBegin;$
// Find data in the tree
// Root is never empty and its value is unique id
$BinaryTreeNode\ current,\ parent = root;$
$\{parent : rcuItr\ \epsilon\ \{\}\}$
$current = parent.Right;$
$\{parent : rcuItr\ \epsilon\ \{Right \mapsto current\}\}$
$\{current : rcuItr\ Right\ \{\}\}$
$while\ (current! = null \&\& current.data! = data)$
{
$\{parent : rcuItr\ (Left|Right)^k\ \{(Left|Right) \mapsto current\}\}$
$\{current : rcuItr\ (Left|Right)^k.(Left|Right)\ \{\}\}$
$if\ (current.data > data)$
{
//if data exists it's in the left subtree
$parent = current;$
$\{parent : rcuItr\ (Left|Right)^k\ \{\}\}$
$\{current : rcuItr\ (Left|Right)^k\ \{\}\}$
$current = parent.Left;$
$\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto current\}\}$
$\{current : rcuItr\ (Left|Right)^k.Left\ \{\}\}$
}
$else\ if\ (current.data < data)$
{
//if data exists it's in the right subtree
$parent = current;$
$\{parent : rcuItr\ (Left|Right)^k\ \{\}\}$
$\{current : rcuItr\ (Left|right)^k\ \{\}\}$
$current = current.Right;$
$\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto current\}\}$
$\{current : rcuItr\ (Left|Right)^k.Right\ \{\}\}$
}
}
$\{parent : rcuItr\ (Left|Right)^k\ \{(Left|Right) \mapsto current\}\}$
$\{current : rcuItr\ (Left|Right)^k.(Left|Right)\ \{\}\}$
// At this point, we've found the node to remove
$BinaryTreeNode\ lmParent = current.Right;$
$BinaryTreeNode\ currentL = current.Left;$
$\{current : rcuItr\ (Left|Right)^k.(Left|Right)\ \{Left \mapsto currentL, Right \mapsto lmParent\}\}$
$\{currentL : rcuItr\ (Left|Right)^k.(Left|Right).Left\ \{\}\}$
$\{lmParent : rcuItr\ Left|Right)^k.(Left|Right).Right\ \{\}\}$

// 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
$if\ (current.Right == null)$
{
$\{parent : rcuItr\ (Left|Right)^k\ \{(Left|Right) \mapsto current\}\}$
$\{current : rcuItr\ (Left|Right)^k.(Left|Right)\ \{Left \mapsto currentL, Right \mapsto null\}\}$
$\{currentL : rcuItr\ (Left|Right)^k.(Left|Right).Left\ \{\}\}$
$if\ (parent.Left == current)$
// parent.Value ¿ current.Value, so make current's left child a left child of parent
$\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto current\}\}$
$\{current : rcuItr\ (Left|Right)^k.Left\ \{Left \mapsto currentL, Right \mapsto null\}\}$
$\{currentL : rcuItr\ (Left|Right)^k.Left.Left\ \{\}\}$
$parent.Left = currentL;$
// parent.Value ¿ current.Value, so make current's left child a left child of parent
$\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto current\}\}$
$\{current : unlinked\}$
$\{currentL : rcuItr\ (Left|Right)^k.Left\ \{\}\}$
$else$
// parent.Value ¡ current.Value, so make current's left child a right child of parent
$\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto current\}\}$
$\{current : rcuItr\ (Left|Right)^k.Right\ \{Left \mapsto currentL, Right \mapsto null\}\}$
$\{currentL : rcuItr\ (Left|Right)^k.Right.Left\ \{\}\}$
$parent.Right = currentL;$
// parent.Value ¿ current.Value, so make current's left child a left child of parent
$\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto current\}\}$
$\{currentL : rcuItr\ (Left|Right)^k.Right\ \{\}\}$
$\{current : unlinked\}$
$SyncStart;$
$SyncStop;$
$\{current : freeable\}$
$Free(current);$
$\{current : undef\}$
}

```
// 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|Right) {Left ↦ null, Right ↦ lmParent}}
 {currentL : rcuItr (Left|Right)^k.(Left|Right).Left {}}
 {lmParent : rcuItr (Left|Right)^k.(Left|Right).Right {}}
 if (parent.Left == current)
  {parent : rcuItr (Left|Right)^k {Left ↦ current}}
  {current : rcuItr (Left|Right)^k.Left {Left ↦ null, Right ↦ lmParent}}
  {lmParent : rcuItr (Left|Right)^k.Left.Right {}}
  // parent.Value ¿ current.Value, so make current's right child a left child of parent
  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.Right {}}
  // parent.Value ¡ current.Value, so make current's right child a right child of parent
  parent.Right = lmParent;
  {parent : rcuItr (Left|Right)^k {Right ↦ lmParent}}
  {lmParent : rcuItr (Left|Right)^k.Right {}}
  {current : unlinked}
  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) {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}
  }
}
```

$else\{$
$\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right\ \{Left \mapsto leftmost\}\}$
$\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left\ \{\}\}$
$while\ (leftmost.Left! = null)$
$\{$
$\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l\ \{Left \mapsto leftmost\}\}$
$\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{\}\}$
$lmParent = leftmost;$
$\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{\}\}$
$\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{\}\}$
$leftmost = lmParent.Left;$
$\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{Left \mapsto leftmost\}\}$
$\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left.Left\ \{\}\}$
$\}$
$currentF.data = leftmost.data;$
$if\ (parent.Left == current)\{$
$\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto current\}\}$
$\{current : rcuItr\ (Left|Right)^k.Left\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$
$\{currentF : rcuFresh\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$
//current's right child a left child of parent
$parent.Left = currentF;$
$\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto currentF\}\}$
$\{current : unlinked\}$
$\{currentF : rcuItr\ (Left|Right)^k.Left\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$
$SyncStart;$
$SyncStop;$
$\{current : freeable\}$
$Free(current);$
$\{current : undef\}$
$\}$
$else\{$
$\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto current\}\}$
$\{current : rcuItr\ (Left|Right)^k.Right\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$
$\{currentF : rcuFresh\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$
//current's right child a right child of parent
$parent.Right = currentF;$
$\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto currentF\}\}$
$\{current : unlinked\}$
$\{currentF : rcuItr\ (Left|Right)^k.Right\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$
$SyncStart;$
$SyncStop;$
$\{current : freeable\}$
$Free(current);$
$\{current : undef\}$
$\}$

$\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l\ \{Left \mapsto leftmost\}\}$
$\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{Left \mapsto null\}\}$
$BinaryTreeNode\ leftmostR = leftmost.Right;$
$\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{Left \mapsto null, Right \mapsto leftmostR\}\}$
$\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l\ \{Left \mapsto leftmost\}\}$
$\{leftmostR : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left.Right\ \{\}\}$
// the parent's left subtree becomes the leftmost's right subtree
$lmParent.Left = leftmostR;$
$\{leftmost : unlinked\}$
$\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l\ \{Left \mapsto leftmostR\}\}$
$\{leftmostR : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{\}\}$
$SyncStart;$
$SyncStop;$
$\{leftmost : freeable\}$
$Free(leftmost);$
$\{leftmost : undef\}$
}
}
$WriteEnd;$
}

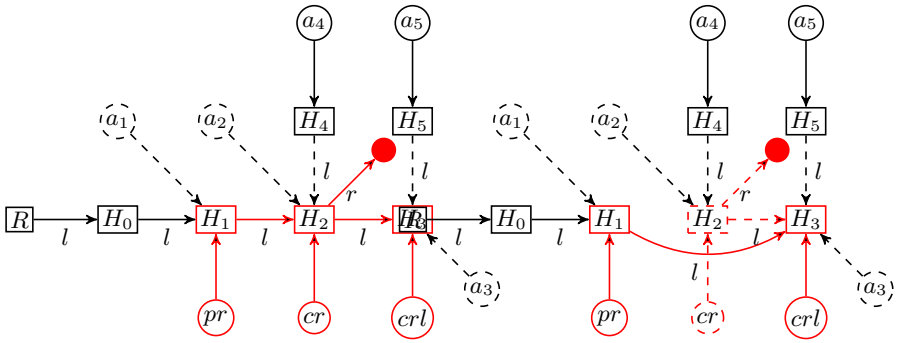
C RCU Bag with Linked-List

```
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;
}
```

$$
\begin{array}{l}
\mathit{void}\ \mathit{remove}\ (\mathit{int}\ \mathit{toDel}\ )\ \{ \\
\quad \mathit{WriteBegin}; \\
\quad \mathit{BagNode}\ \mathit{current},\ \mathit{parent}\ =\ \mathit{head}; \\
\quad \mathit{current}\ =\ \mathit{parent}.\mathit{Next}; \\
\quad \{\mathit{current} : \mathit{rcuItr}\ \mathit{Next}\ \{\}\} \\
\quad \{\mathit{parent} : \mathit{rcuItr}\ \epsilon\ \{\mathit{Next} \mapsto \mathit{current}\}\} \\
\quad \mathit{while}\ (\mathit{current}.\mathit{Next}! = \mathit{null}\&\&\mathit{current}.\mathit{data} \neq \mathit{toDel})\ \{ \\
\quad\quad \left\{\mathit{parent} : \mathit{rcuItr}\ (\mathit{Next})^k\ \{\mathit{Next} \mapsto \mathit{current}\}\right\} \\
\quad\quad \left\{\mathit{current} : \mathit{rcuItr}\ \mathit{Next}.(\mathit{Next})^k.\mathit{Next}\ \{\}\right\} \\
\quad\quad \mathit{parent} = \mathit{current}; \\
\quad\quad \left\{\mathit{current} : \mathit{rcuItr}\ \mathit{Next}.(\mathit{Next})^k.\mathit{Next}\ \{\}\right\} \\
\quad\quad \left\{\mathit{parent} : \mathit{rcuItr}\ \mathit{Next}.(\mathit{Next})^k.\mathit{Next}\ \{\}\right\} \\
\quad\quad \mathit{current} = \mathit{parent}.\mathit{Next}; \\
\quad\quad \left\{\mathit{parent} : \mathit{rcuItr}\ \mathit{Next}.(\mathit{Next})^k.\mathit{Next}\ \{\mathit{Next} \mapsto \mathit{current}\}\right\} \\
\quad\quad \left\{\mathit{current} : \mathit{rcuItr}\ \mathit{Next}.(\mathit{Next})^k.\mathit{Next}.\mathit{Next}\ \{\}\right\} \\
\quad \} \\
\quad //\text{We don't need to be precise on whether next of current is null or not} \\
\quad \left\{\mathit{parent} : \mathit{rcuItr}\ \mathit{Next}.(\mathit{Next})^k.\mathit{Next}\ \{\mathit{Next} \mapsto \mathit{current}\}\right\} \\
\quad \left\{\mathit{current} : \mathit{rcuItr}\ \mathit{Next}.(\mathit{Next})^k.\mathit{Next}.\mathit{Next}.\mathit{Next}\ \{\mathit{Next} \mapsto \mathit{null}\}\right\} \\
\quad \mathit{BagNode}\ \mathit{currentL} = \mathit{current}.\mathit{Next}; \\
\quad \left\{\mathit{parent} : \mathit{rcuItr}\ \mathit{Next}.(\mathit{Next})^k.\mathit{Next}\ \{\mathit{Next} \mapsto \mathit{itr}\}\right\} \\
\quad \left\{\mathit{currentL} : \mathit{rcuItr}\ \mathit{Next}.(\mathit{Next})^k.\mathit{Next}.\mathit{Next}.\mathit{Next}\ \{\}\right\} \\
\quad \left\{\mathit{current} : \mathit{rcuItr}\ \mathit{Next}.(\mathit{Next})^k.\mathit{Next}.\mathit{Next}\ \{\mathit{Next} \mapsto \mathit{currentL}\}\right\} \\
\quad \mathit{current}.\mathit{Next} = \mathit{currentL}; \\
\quad \left\{\mathit{parent} : \mathit{rcuItr}\ \mathit{Next}.(\mathit{Next})^k.\mathit{Next}\ \{\mathit{Next} \mapsto \mathit{itrN}\}\right\} \\
\quad \left\{\mathit{currentL} : \mathit{rcuItr}\ \mathit{Next}.(\mathit{Next})^k.\mathit{Next}.\mathit{Next}\ \{\}\right\} \\
\quad \{\mathit{current} : \mathit{unlnked}\} \\
\quad \mathit{SyncStart}; \\
\quad \mathit{SyncStop}; \\
\quad \{\mathit{current} : \mathit{freeable}\} \\
\quad \mathit{Free}(\mathit{current}); \\
\quad \{\mathit{current} : \mathit{undef}\} \\
\quad \mathit{WriteEnd}; \\
\}
\end{array}
$$

$$
\begin{array}{l}
void\ add(int to Add)\{ \\
\quad WriteBegin; \\
\quad BagNode\ nw = new; \\
\quad nw.data = toAdd; \\
\quad \{nw : rcuFresh\ \{\}\} \\
\quad BagNode\ current,\ parent\ =\ head; \\
\quad parent.Next\ =\ current; \\
\quad \{current : rcuItr\ Next\ \{\}\} \\
\quad \{parent : rcuItr\ \epsilon\ \{Next \mapsto current\}\} \\
\quad while\ (current.Next! = null)\ \{ \\
\quad\quad \left\{parent : rcuItr\ (Next)^k\ \{Next \mapsto current\}\right\} \\
\quad\quad \left\{current : rcuItr\ Next.(Next)^k.Next\ \{\}\right\} \\
\quad\quad parent = current; \\
\quad\quad current = parent.Next; \\
\quad\quad \left\{parent : rcuItr\ (Next)^k.Next\ \{Next \mapsto current\}\right\} \\
\quad\quad \left\{current : rcuItr\ Next.(Next)^k.Next.Next\ \{\}\right\} \\
\quad \} \\
\quad \left\{parent : rcuItr\ (Next)^k.Next\ \{Next \mapsto current\}\right\} \\
\quad \left\{current : rcuItr\ Next.(Next)^k.Next.Next\ \{Next \mapsto null\}\right\} \\
\quad nw.next = null; \\
\quad \{nw : rcuFresh\ \{Next \mapsto null\}\} \\
\quad current.Next = nw \\
\quad \left\{parent : rcuItr\ (Next)^k.Next\ \{Next \mapsto nw\}\right\} \\
\quad \left\{current : rcuItr\ (Next)^k.Next.Next\ \{Next \mapsto nw\}\right\} \\
\quad \left\{nw : rcuItr\ Next.(Next)^k.Next.Next.Next\ \{Next \mapsto null\}\right\} \\
\quad WriteEnd; \\
\}
\end{array}
$$

D RCU Unlinking and Fresh Linking



(a) Framing before unlinking the heap node pointed by `current`-cr.

(b) Safe unlinking of the heap node pointed by `current`-cr via Framing

Fig. 30: Safe unlinking a heap node from a BST

(***Mutation - Unlinking a Heap Node***) - 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 effect of the mutation, we illustrate the setting for unlinking a heap node including resources such as stack pointer aliases in Figures 30a and 30b. 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 foot print 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 `currenL`-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 which depicts `null`, the l field which depicts $Left$ and r which depicts $Right$ field.

The type rule for unlinking must assert the "proper linkage" in between the heap nodes involved in unlinking. We see the proper linkage relation between in Figure 30a 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 $\mathcal{N}([f_1 \rightharpoonup z])$ $(Left \mapsto current)$ of x, $\mathcal{N}'([f_2 \rightharpoonup r])$ $(Left \mapsto currentL)$ of y. In accordance with the field mappings, the type rule also asserts that x has the path ρ $((Left)^k)$, 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 aliasing 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 30a and 30b. 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

$$(\neg\mathsf{MayAlias}(\rho''', \{\rho, \rho', \rho''\}))$$

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(ρ''') 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, ρ''', referring to heap nodes deeper than memory footprint

$$(\forall_{\rho''''\neq\epsilon}.\, \neg\mathsf{MayAlias}(\rho''', \rho''.\rho''''))$$

The reason for framing-out these dashed path aliases is obvious when we look at the changes from the Figure 30a to Figure 30b. For example, a_1 points to H_1 which has object field $Next\text{-}n$ pointing to H_2 which is also pointed by `current` as depicted in the Figure 30a. However, in Figure 30b, n of H_1 is pointing to H_3 though a_1 still points to H_1. This change invalidates the field mapping $Left \mapsto current$ 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 30a, both `current` and a_2 are in the rcultr type and point to H_2. After the unlinking action, the type of `current` becomes `unlinked` although a_2 is still in the `rcuItr` 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 happens, both `currentL` and a_3 have the same path components. After unlinking, the path of `currentL`-crl gets shortened as the path to heap node it points, H_3, changes to $(Left)^k.Left$. 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 alising examples in Figures 30a and 30b: 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 30a and 30b.

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 30b.

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 `null` via

$$\forall_{f \in dom(\mathcal{N}')}.\, f \neq f_2 \implies (\mathcal{N}'(f) = \mathsf{null})$$

E Types and Rules for RCU Read Section

$\tau_{local} := \mathsf{rcultr}$ $\quad\boxed{\Gamma \vdash_R \alpha \dashv \Gamma'}$

(T-Root)
$$\frac{y \notin \mathsf{FV}(\Gamma)}{\Gamma, r : \mathsf{rcuRoot}, y : \mathsf{undef} \vdash y = r \dashv y : \mathsf{rcultr}, r : \mathsf{rcuRoot}, \Gamma}$$

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

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

(ToRCURead)
$$\frac{\Gamma, y : \mathsf{rcultr} \vdash_R \bar{s} \dashv \Gamma' \quad \mathsf{FType}(f) = \mathsf{RCU}}{\Gamma \vdash \mathsf{RCURead}\, x.f \text{ as } y \text{ in } \{\bar{s}\}} \qquad \boxed{\Gamma \vdash_{M,R} \bar{s} \dashv \Gamma'}$$

(T-Loop1)
$$\frac{\Gamma(x) = \mathsf{bool} \qquad \Gamma \vdash \bar{s} \dashv \Gamma}{\Gamma \vdash \mathsf{while}(x)\{\bar{s}\} \dashv \Gamma}$$

(T-Branch2)
$$\frac{\Gamma(x) = \mathsf{bool} \qquad \Gamma \vdash \bar{s}_1 \dashv \Gamma' \qquad \Gamma \vdash \bar{s}_2 \dashv \Gamma'}{\Gamma \vdash \mathsf{if}(x) \text{ then } \bar{s}_1 \text{ else } \bar{s}_2 \dashv \Gamma'}$$

(T-Seq)
$$\frac{\Gamma_1 \vdash \bar{s_1} \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{M,R} \bar{s_2} \dashv \Gamma_3}{\Gamma_1 \vdash_{M,R} \bar{s_1} \; ; \; \bar{s_2} \dashv \Gamma_3}$$

(T-Par)
$$\frac{\Gamma_1 \vdash_R \bar{s_1} \dashv \Gamma_1' \qquad \Gamma_2 \vdash_{M,R} \bar{s_2} \dashv \Gamma_2'}{\Gamma_1, \; \Gamma_2 \vdash \bar{s_1} || \bar{s_2} \dashv \Gamma_1', \Gamma_2'}$$

(T-Exchange)
$$\frac{\Gamma, y : T', x : T, \Gamma' \vdash \bar{s} \dashv \Gamma''}{\Gamma, x : T, y : T', \Gamma' \vdash \bar{s} \dashv \Gamma''}$$

(T-Conseq)
$$\frac{\Gamma \prec: \Gamma' \qquad \Gamma' \vdash \bar{s} \dashv \Gamma'' \qquad \Gamma'' \prec: \Gamma'''}{\Gamma \vdash \bar{s} \dashv \Gamma'''}$$

(T-Skip)
$$\frac{}{\Gamma \vdash_{M,R} \mathsf{skip} \dashv \Gamma}$$

Fig. 31: Type Rules for Read critical section for RCU Programming and standard type rules