Safe Deferred Memory Reclamation with Types

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

Drexel University
{ik335,csgordon}@drexel.edu

Abstract. Memory management in lock-free data structures remains a major challenge in concurrent programming. Design techniques including read-copy-update (RCU) and hazard pointers provide workable solutions, and are widely used to great effect. These techniques rely on the concept of a grace period: nodes that should be freed are placed on a *deferred* free list, and all threads obey a protocol to ensure that the deallocating thread can detect when all possible readers have completed their use of the object. This provides an approach to safe deallocation, but only when these subtle protocols are implemented correctly.
We present a static type system to ensure correct use of RCU memory management: that nodes removed from a data structure are always scheduled for subsequent deallocation, and that nodes are scheduled for deallocation at most once. As part of our soundness proof, we give an abstract semantics for RCU memory management primitives which captures the fundamental properties of RCU. Our type system allows us to give the first proofs of memory safety for RCU linked list and binary search tree implementations without requiring full verification.

1 Introduction

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

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

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

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

Our contributions include:

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

2 Background & Motivation

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

As with concrete RCU implementations, we assume threads operating on a structure are either performing read-only traversals of the structure — *reader*

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

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

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

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.

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 **??** gives the code for adding and removing nodes from a bag. Type checking for all code, including membership queries for bag, can be found in our technical report [**?**] Appendix **??**. 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

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

hold references to unlinked nodes to finish their critical sections. `SyncStop` blocks execution of the writer thread until these readers exit their read critical section (via `ReadEnd`). These are the essential primitives for the implementation of an RCU data structure.

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

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

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

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

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

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

- Reader threads never modify the structure
- No thread holds an interior pointer into the RCU structure across critical sections

- Unlinked nodes are always freed by the unlinking thread *after* the unlinking, *after* a grace period, and *inside* the critical section
- Nodes are freed at most once

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

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

and our type system in Section **??** guarantees these invariants.

3 Semantics

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

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

The lock l enforces mutual exclusion between write-side critical sections. The root location rt is the root of an RCU data structure. We model only a single global RCU data structure, 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 **??** gives operational semantics for *atomic* actions; conditionals, loops, and sequencing all have standard semantics, and parallel composition uses sequentially-consistent interleaving semantics.

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

Standard RCU APIs include a primitive `synchronize_rcu()` to wait for a grace period for the current readers. We decompose this here into two actions, `SyncStart` and `SyncStop`. `SyncStart` initializes the blocking set to the current set of readers — the threads that may have already observed any nodes the writer

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

$$[\![\mathtt{x.f=y}]\!](s,h,l,R,F) \ \Downarrow_{\text{tid}} \ \triangleq (s, h[s(x,tid), f \mapsto s(y,tid)], l, R, F) \ \ (\text{HU}\textsc{pdate})$$

$$[\![\mathtt{y=x.f}]\!](s,h,l,R,F) \ \Downarrow_{\text{tid}} \ \triangleq ((s[(y,tid) \mapsto h(s(x,tid),f)], h, l, R, F) \ \ (\text{HR}\textsc{ead})$$

$$[\![\mathtt{y=x}]\!](s,h,l,R,F) \ \Downarrow_{\text{tid}} \ \triangleq (s[(y,tid) \mapsto (x,tid)], h, l, R, F) \ \ (\text{SU}\textsc{pdate})$$

$$[\![\mathtt{y=new}]\!](s,h,l,R,F) \ \Downarrow_{\text{tid}} \ \triangleq (s,h,l,R,F) \ \ (\text{HA}\textsc{llocate})$$

$$\begin{array}{l} [\![\mathtt{asyncDelayedFree}(x)]\!](s,h,l,R,F) \ \Downarrow_{\text{tid}} \ \triangleq (s,h,l,R,F \uplus \{(R,S(x))\}) \\ \quad \mathtt{provided} \ \ s[(y,tid) \mapsto o], \ h[o \mapsto \mathtt{new}] \text{ and } \mathtt{default}(\mathsf{FType}(f)) \qquad (\text{AF}\textsc{ree}) \\ \qquad \lambda o', f. \ \text{if } o = o' \ \text{ skip } \ \text{else } h(o',f) \end{array}$$

$$[\![\mathtt{gcStep}]\!](s,h,l,R,F) \ \Downarrow_{\text{tid}} \ \triangleq (s,h',l,R,F) \ \ (\text{GC})$$

$$\begin{array}{l} [\![\mathtt{WriteBegin}]\!](s,h,\text{unlocked},R,F) \ \Downarrow_{\text{tid}} \ \triangleq (s,h,\text{tid},R,F) \\ \mathtt{provided} \ \ s[(x,tid) \mapsto o], \ (\forall f, o'.\, o \neq o' \Rightarrow h(o',f) = h'(o',f))) \ (\text{RCU-W-B}\textsc{egin}) \\ \qquad \text{and } \forall f.\, h'(o,f) \ \ \mathsf{undefined} \end{array}$$

$$[\![\mathtt{WriteEnd}]\!](s,h,\text{tid},R,F) \ \Downarrow_{\text{tid}} \ \triangleq (s,h,\mathsf{unlocked},R,F) \qquad \text{provided } tid \notin R \ (\text{RCU-W-E}\textsc{nd})$$

$$[\![\mathtt{ReadBegin}]\!](s,h,l,R,F) \ \Downarrow_{\text{tid}} \ \triangleq (s,h,l,R \uplus \{\text{tid}\},F) \qquad \text{provided } tid \neq l \ (\text{RCU-R-B}\textsc{egin})$$

$$[\![\mathtt{ReadEnd}]\!](s,h,l,R \uplus \{\text{tid}\},F) \ \Downarrow_{\text{tid}} \ \triangleq (s,h,l,R,F \setminus \text{tid}) \, \text{provided } \ F \setminus tid \ \text{ as } \ \{(T \setminus tid\}, o) \mid (T,o) \in F\} \ (\text{RCU-R-E}\textsc{n}$$

Fig. 2: Operational Semantics for RCU Programming Model

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

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

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

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

4 Type System & Programming Language

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

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

it simplifies reasoning (by developers or a type system) about when a node has become unreachable in the heap.

We also demonstrate that the type system is not only sound, but useful: we show how it types Figure **??**'s list-based bag implementation [**?**]. We also give type checked fragments of a binary search tree to motivate advanced features of the type system; the full typing derivation can be found in our technical report [**?**] Appendix **??**. 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 **??** introduces RCU types and the need for subtyping. Section **??**, shows how types describe program states, through code for Figure **??**'s list-based bag example. Section **??** introduces the type system itself.

RCU Types There are six types used in Write critical sections

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3: Subtyping rules.

location referenced by an unlinked reference may still be accessed by reader threads, which may have acquired their own references before the node became unreachable. Newly-arrived readers, however, will be unable to gain access to these referents.

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

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

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

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

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

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

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

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

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

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

Fig. 4: Type rules for control-flow.

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

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

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

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

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

Fig. 5: Choosing fields to read.

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

4.2 Types in Action

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

Below, we explain our types through the list-based bag implementation [**?**] from Figure **??**, highlighting how the type rules handle different parts of the code. Figure **??** 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 **??** begins by reading the root into a new rcultr reference used to begin traversing the structure. After each initial read (line 12 of `add` and line 4 of `remove`), the path of `cur` reference is the empty path (ϵ) and the field map is empty ($\{\}$), because it is an alias to the root, and none of its field contents are known yet.

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

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

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

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

4.3 Type Rules

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

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

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

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

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

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

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

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

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

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

(T-WriteFH)

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

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

(T-UnlinkH)

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

(T-Replace)

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

(T-Insert)

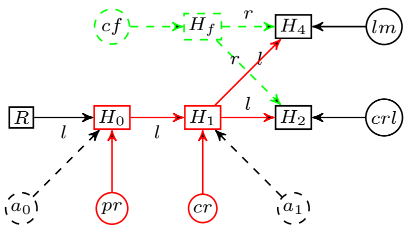
$$\frac{\begin{array}{c} \mathcal{N}' = \mathcal{N}([f \rightharpoonup o \setminus n]) \qquad \rho.f = \rho_1 \qquad \rho_1.f_4 = \rho_2 \\ \mathcal{N}(f) = \mathcal{N}_1(f_4) \qquad \forall_{f_2 \in dom(\mathcal{N}_1)} .\ f_4 \neq f_2 \implies \mathcal{N}_1(f_2) = \mathsf{null} \qquad \mathsf{FV}(\Gamma)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

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

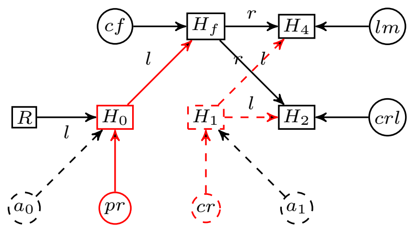
Fig. 6: Type rules for write side critical section.

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

Replacing with a Fresh Node Replacing with a rcuFresh reference faces the same aliasing complications as direct unlinking. We illustrate these challenges in Figures **??** and **??**. Our technical report [?] also includes Figures **??** and **??** in Appendix **??** to illustrate complexities in unlinking. The square R nodes are



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



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

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

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 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 **??**) 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 replacing with a fresh node to just one fresh reference and the object it points to. Otherwise another mechanism would be required to ensure that once a fresh reference was linked into the heap, there were no aliases still typed as fresh — since that would have risked linking the same reference into the heap in two locations.

The transition from the Figure **??** to **??** illustrates the effects of the heap mutation (replacing with a fresh node). The reasoning in the type system for replacing with a fresh node is nearly the same as for unlinking an existing node, with one exception. In replacing with a fresh node, there is no need to consider the paths of nodes deeper in the tree than the point of mutation. In the unlinking case, those nodes' static paths would become invalid. In the case of replacing with a fresh node, those descendants' paths are preserved. Our type rule for ensuring safe replacement (T-Replace) prevents path aliasing (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-Replace).

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

There is also another rule, T-LinkF-Null, not shown in Figure **??**, 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***) We introduce the *syntactic sugaring* RCUWrite $x.f$ as y in $\{C\}$ for write-side critical sections where the analogous syntactic sugaring can be found for read-side critical sections in Appendix **??** of the technical report [**?**].

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

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

5 Evaluation

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

Figure **??** gives the type-annotated code for `add` and `remove` operations on a linked list implementation of a bag data structure, following McKenney's example [**?**]. Appendix **??** of the technical report [**?**] 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 [**?**]: their code supports fine-grained locking for multiple writers, while ours supports only one writer by virtue of using our single-writer primitives. For lack of space the annotated code is only in Appendix **??** of the technical report [**?**], but it motivates some of the conditional-related flexibility discussed in Section **??**. The use of disjunction $(Left|Right)$ in field maps and paths is required to capture traversals which follow different fields at different times, such as the lookup in a binary search tree.

The most subtle aspect of the deletion is the final step in the case the node H_1 to remove has both children. In this case, the value H_s of the left-most node of H_1's right child — the next element in the collection order — is copied into a new *freshly-allocated* node as shown in Figure **??**, which is then used to *replace* node H_1 as shown in Figure **??**: the replacement's fields exactly match H_1's except for the data (T-Replace via $\mathcal{N}_1 = \mathcal{N}_2$) as showin in Figure **??**, and the parent is updated to reference the replacement, unlinking H_1. At this point, as shown in Figures **??**-**??**, there are two nodes with value H_s in the tree (*weak* BST property of the Citrus [**?**]): the replacement node, and what was the left-most node under H_1's right child. This latter (original) node for H_s must be unlinked as shown in Figure **??**, which is simplified because by being left-most the left child is null, avoiding another round of replacement (T-UnlinkH via $\forall_{f \in dom(\mathcal{N}_1)}. f \neq f_2 \implies (\mathcal{N}_1(f) = \mathsf{null})$.

The complexity in checking safety here is that once H_1 is found after traversing the subtree T_0 with references

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

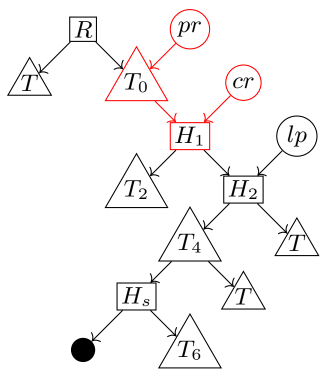
where T_0 traversal is summarized as $(l|k)^k$, another loop is used to find H_s and its parent (since that node will later be removed as well) after traversing the subtree T_4 with references

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

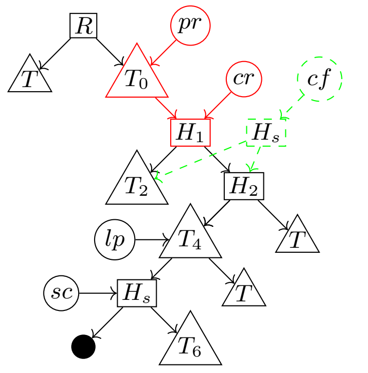
where T_4 traversal is summarized as $(l|m)^m$.

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

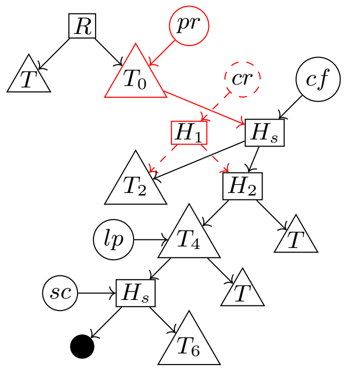
To the best of our knowledge, we are the first to check such code for memory-safe use of RCU primitives modularly, without appeal to the specific implementation of RCU primitives.



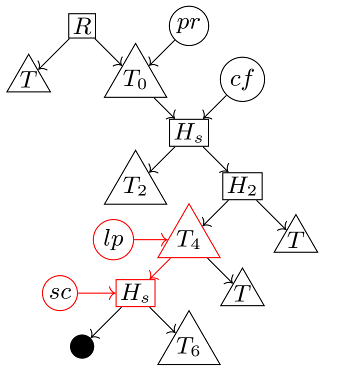
(a) The writer traverses subtree T_0 to find the heap node H_1 with local references *pr* and *cr*.



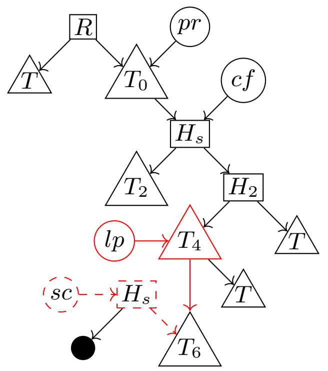
(b) Traverse subtree T_4 starting from H_2 with references *lp* and *sc* to find successor H_1' of H_1. Duplicating H_1' as a fresh heap node before replacing H_1 with the fresh one.



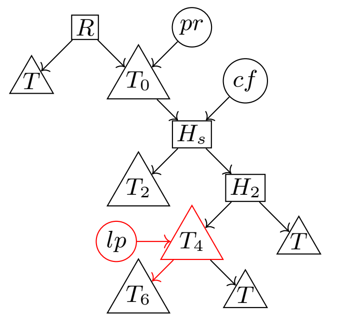
(c) Replace H_1 with fresh successor and synchronize with the readers.



(d) Unlinks old successor referenced by *sc*.



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



(f) Reclamation of the old successor.

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

6 Soundness

This section outlines the proof of type soundness – our full proof appears in Appendices **??**, **??**, **??** and **??** of the technical report [**?**]. We prove type soundness by embedding the type system into an abstract concurrent separation logic called the Views Framework [**?**], which when given certain information about proofs for a specific language (primitives and primitive typing) gives back a full program logic including choice and iteration. As with other work taking this approach [**?**,**?**], this consists of several key steps:

1. Define runtime states and semantics for the atomic actions of the language. These are exactly the semantics from Figure **??** in Section **??**
2. Define a denotational interpretation $[\![-]\!]$ of the types (Figure **??**) in terms of an *instrumented* execution state – a runtime state (Section **??**) with additional bookkeeping to simplify proofs. The denotation encodes invariants specific to each type, like the fact that unlinked references are unreachable from the heap. The instrumented execution states are also constrained by additional *global* WellFormedness *invariants* – see Appendix **??** of the technical report [**?**] for detailed formal invariants – the type system is intended to maintain, such as tree structure of the data structure.
3. Prove a lemma – called *Axiom Soundness* (Lemma **??**) – that the type rules for atomic actions are sound. Specifically, that given a state in the denotation of the pre-type-environment of a primitive type rule, the operational semantics produce a state in the denotation of the post-type-environment. This includes preservation of global invariants.
4. Give a desugaring $\downarrow - \downarrow$ of non-trivial control constructs (Figure **??**) into the simpler non-deterministic versions provided by Views.

The top-level soundness claim then requires proving that every valid source typing derivation corresponds to a valid derivation in the Views logic: $\forall \Gamma, C, \Gamma', \Gamma \vdash_M C \dashv \Gamma' \Rightarrow \{[\![\Gamma]\!]\} \downarrow C \downarrow \{[\![\Gamma']\!]\}$. Because the parameters given to the Views framework ensure the Views logic's Hoare triples $\{-\}C\{-\}$ are sound, this proves soundness of the type rules with respect to type denotations. Because our denotation of types encodes the property that the post-environment of any type rule accurately characterizes which memory is linked vs. unlinked, etc., and the global invariants ensure all allocated heap memory is reachable from the root or from some thread's stack, this entails that our type system prevents memory leaks.

6.1 Proof

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

Logical State Section **??** defined what Views calls *atomic actions* (the primitive operations) and their semantics of runtime *machine states*. The Views

Framework uses a separate notion of instrumented (logical) state over which the logic is built, related by a concretization function $\lfloor - \rfloor$ taking an instrumented state to the machine states of Section **??**. Most often — including in our proof — the logical state adds useful auxiliary state to the machine state, and the concretization is simply projection. Thus we define our logical states LState as:

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

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

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

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

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

Invariants of RCU Views and Denotations of Types In this section we aim to convey the intuition behind the predicate WellFormed which enforces global invariants on logical states, and how it interacts with the denotations of types in key ways. WellFormed is the conjunction of a number of more specific invariants, which we outline here. For full details, see Appendix **??** of the technical report [**?**].

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

Invariants on Grace-Period Our logical state (Section **??**) includes some "free list" auxiliary state tracking which readers are still accessing *each* unlinked node during grace periods. This must be consistent with the bounding thread set B in the machine state. The Readers-In-Free-List invariant asserts that all reader threads with observations of unlinked locations are in the to-free lists for those locations. This is essentially tracking which readers are being "shown grace" for each location. The $\mathsf{Iterators\text{-}Free\text{-}List}$ invariant complements this by asserting all readers with such observations are in the bounding thread set.

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

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

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

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

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

Assertions in the Views logic are (almost) sets of the logical states that satisfy a validity predicate $\mathsf{WellFormed}$, outlined in Section **??**:

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

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

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

Fig. 9: Type Environments

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

$$\llbracket - \rrbracket_{\_} : \mathsf{TypeEnv} \to \mathsf{TID} \to \mathcal{P}(\mathcal{M})$$

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

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

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

Composition and Interference To support framing (weakening), the Views Framework requires that views form a partial commutative monoid under an operation $\bullet : \mathcal{M} \longrightarrow \mathcal{M} \longrightarrow \mathcal{M}$, provided as a parameter to the framework. The framework also requires an interference relation $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$ between views to reason about local updates to one view preserving validity of adjacent views (akin to the small-footprint property of separation logic). Figure **??** defines our composition operator and the core interference relation $\mathcal{R}_0$ — the actual inferference between views (between threads, or between a local action and framed-away state) is the reflexive transitive closure of $\mathcal{R}_0$. Composition is mostly straightforward point-wise union (threads' views may overlap) of each component. Interference bounds the interference writers and readers may inflict on each other. Notably, if a view contains the writer thread, other threads may

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

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

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

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

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

not modify the shared portion of the heap, or release the writer lock. Other aspects of interference are natural restrictions like that threads may not modify each others' local variables. WellFormed states are closed under both composition (with another WellFormed state) and interference ($\mathcal{R}$ relates WellFormed states only to other WellFormed states).

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

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

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

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

Additionally, interference must distribute over composition:

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

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

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

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

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

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

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

Proof. In Appendix **??** Lemma **??** of the technical report [**?**].

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

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

There are two remaining subtleties to address. First, commands C also require translation: the Views Framework has only non-deterministic branches and loops, so the standard versions from our core language must be encoded. The approach to this is based on a standard idea in verification, which we show here for conditionals as shown in Figure **??**. $\mathsf{assume}(b)$ is a standard idea in verification semantics [**?**,**?**], which "does nothing" (freezes) if the condition b is false, so its postcondition in the Views logic can reflect the truth of b. assume in Figure **??** adapts this for the Views Framework as in other Views-based proofs [**?**,**?**], specifying sets of machine states as a predicate. We write boolean expressions as shorthand for the set of machine states making that expression true.

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

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

We are now finally ready to prove the key lemmas of the soundness proof, relating subtying to view shifts, proving soundness of the primitive actions, and

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

finally for the full type system. These proofs occur once for the writer type system, and once for the reader; we show here only the (more complex) writer obligations:

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

Proof. By case analysis on α. Details in Appendix **??** of the techical report [**?**].

Lemma 3 (Context-SubTyping-M). $\Gamma \prec: \Gamma' \implies [\![\Gamma]\!]_{M,tid} \sqsubseteq [\![\Gamma']\!]_{M,tid}$

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

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

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

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

7 Related Work

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

Modeling RCU and Memory Models Alglave et al. [**?**] 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 [**?**] satisfies this property. According to the fundamental requirements of RCU [**?**], our model in Section **??** can be considered as a valid RCU implementation satisfying all requirements for an RCU implementation(assuming sequential consistency) aside from one performance optimization, *Read-to-Write Upgrade*, which is important in practice but not memory-safety centric – see the technical report [**?**] for detailed discussion on satisfying RCU requirements.

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

- *Publish-Subscribe Guarantee*: Fresh heap nodes cannot be observed by the reader threads until they are published. As we see in the operational semantics, once a new heap location is allocated it can only be referenced by a local variable of type fresh. Once published, the local type for that reference becomes rcultr, indicating it is now safe for the reader thread to access it with local references in rcultr type. We discuss the related type assertions for inserting/replacing(Figures **??**-**??**) a fresh node in Section **??** and the related invariants in Section **??**.
- *RCU Primitives Guaranteed to Execute Unconditionally*: Unconditional execution of RCU Primitives are provided by the definitions in our operational semantics for our RCU primitives(e.g. `ReadBegin`, `ReadEnd`, `WriteBegin` and `WriteEnd`) as their executions do not consider failure/retry.
- *Guaranteed Read-to-Write Upgrade*: This is a performance optimization which allows the reader threads to upgrade the read-side critical section to the write-critical section by acquiring the lock after a traversal for a data element and ensures that the upgrading-reader thread exit the read-critical section before calling RCU synchronization. This optimization also allows sharing the traversal code between the critical sections. *Read-to-Write* is an important optimization in practice but largely orthogonal to memory-safety. Current version of our system provides a strict separation of *traverse-and-update* and *traverse-only* intentions through the type system(e.g. different iterator types and rules for the RCU Write/Read critical sections) and the programming primitives. As a future work, we want to extend our system to support this performance optimization.

To the best of our knowledge, ours is the first abstract *operational* model for a Linux kernel-style RCU implementation – others are implementation-specific [**?**] or axiomatic like Alglave et al.'s.

Tassarotti et al. model a well-known way of implementing RCU synchronization without hurting readers' performance, Quiescent State Based Reclamation(QSBR) [**?**] where synchronization between the writer thread and reader threads provided via per-thread counters. Tassarotti et al. [**?**] uses a protocol based program logic based on separation and ghost variables called GPS [**?**] to verify a user-level implementation of RCU with a singly linked list client under *release-acquire* semantics, which is a weaker memory model than sequential-consistency. They require *release-writes* and *acquire-reads* to the QSRB counters for proper synchronization in between the mutator and the reader threads. This protocol is exactly what we enforce over the logical observations of the mutator thread: from unlinked to freeable. Tassarotti et al.'s synchronization for linking/publishing new nodes occurs in a similar way to ours, so we anticipate it would be possible to extend our type system in the future for similar weak memory models.

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

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

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

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

Type Systems Howard et al. [?,?] present a Haskell library called *Monadic RP* which provides types and relativistic programming constructs for write/read critical sections which enforce correct usage of relativistic programming pattern.

They also have only checked a linked list. They claim handling trees (look-up followed by update) as a future work [?]. Thus our work is the first type system for ensuring correct use of RCU primitives that is known to handle more complex structures than linked lists.

8 Conclusions

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

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

A Complete Soundness Proof of Atoms and Structural Program Statements

A.1 Complete Constructions for Views

To prove soundness we use the Views Framework [?]. 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 **??** in Section **??** and type rule in Figures **??** Section **??** and **??** Appendix **??**, 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)

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

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.

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

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

Fig. 12: 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 **??**. 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 **??** An important property of composition is that it preserves validity of logical states:

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

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

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

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

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

Fig. 13: Composition($\bullet$) and Thread Interference Relation($\mathcal{R}_0$)

Lemma 5 (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 $\mathsf{Wellformed}$(x) and $\mathsf{Wellformed}$(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 $x \bullet y = z$, $\mathsf{Wellformed}$(z) holds. Both x and y have components $((s_x, h_x, l_x, rt_x, R_x, B_x), O_x, U_x, T_x, F_x)$ 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:

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

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

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

Thread interference relation

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

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

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

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

Lemma 6 (Valid $\mathcal{R}_0$ Interference). *For any m and m', if* $\mathsf{WellFormed}(m)$ *and $m\mathcal{R}_0m'$, then* $\mathsf{WellFormed}(m')$*.*

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

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

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

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

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

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 7 (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

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

is stable, and must show that

$$[\![\Gamma']\!]_{\mathsf{M},tid} \cap [\![x : \tau]\!]_{tid}$$

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

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

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

Cases in which denotations, $[\![x : T]\!]$, 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:

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

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

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:

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

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

prove this case.

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

$$\mathcal{R}([\![\Gamma]\!]_{\mathsf{R},tid}) \subseteq [\![\Gamma]\!]_{\mathsf{R},tid}$$

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

Proof. Proof is similar to one for Lemma **??** where there is only one simple case, $[\![x : \mathsf{rcultr}]\!]$.

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:

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

The antecedent of the implication is a type judgment(shown in Figure **??** Section **??**, Figure **??** Section **??** and Figure **??** Appendix **??**) 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 **??**. $\mathsf{assume}(b)$ is a standard construct in verification semantics [**?**] [**?**], 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 [**?**,**?**].

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

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

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

Fig. 14: Encoding of assume(b)

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

Lemma 9 (Axiom of Soundness for Atoms). *For each axiom,* $\Gamma_1 \vdash_{\mathsf{RMO}} \alpha \dashv \Gamma_2$*, 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. All the cases proved as different lemmas in Section **??**.

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

Lemma 10 (Views Embedding for Read-Side).

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

Proof. Proof is similar to the one for Lemma **??** 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 11 (Views Embedding for Write-Side).

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

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: $\mathsf{rcultr}\,\rho\mathcal{N}$, undef, $\mathsf{rcuFresh}$, $\mathsf{unlinked}$ and $\mathsf{freeable}$. 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 12 (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.

A.2 Complete Memory Axioms

1. Ownership invariant in Figure ?? 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 ?? 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 ?? asserts that the unique root location can only be aliased with thread local references through which the unique root location is observed as iterator.
4. Iterators-Free-List invariant in Figure ?? 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.
5. Unlinked-Reachability invariant in Figure ?? 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.

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

Fig. 15: Ownership

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

Fig. 16: Reader-Writer-Iterator-Coexistence-Ownership

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

Fig. 17: Alias with Unique Root

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

Fig. 18: Iterators-Free-List

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

Fig. 19: Unlinked-Reachability

6. Free-List-Reachability invariant in Figure **??** 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.
7. Writer-Unlink invariant in Figure **??** asserts that the writer thread cannot observe a heap location as unlinked.

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

Fig. 20: Free-List-Reachability

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

Fig. 21: Writer-Unlink

8. Fresh-Reachable invariant in Figure **??** 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.

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

Fig. 22: Fresh-Reachable

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

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

Fig. 23: Fresh-Writer

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

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

Fig. 24: Fresh-Not-Reader

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

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

Fig. 25: Fresh-Points-Iterator

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

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

Fig. 26: Writer-Not-Reader

13. Readers-Iterator-Only invariant in the Figure **??** asserts that a reader threads can only make $\mathsf{iterator}$ observation on a heap location.

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

Fig. 27: Readers-Iterator-Only

14. Readers-In-Free-List invariant in Figure **??** asserts that for any mapping from a location to a set of threads in the free list we know the fact that this set of threads is a subset of bounding threads(which itself is subset of reader threads).

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

Fig. 28: Readers-In-Free-List

15. Heap-Domain invariant in the Figure **??** 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. 29: Heap-Domain

16. Unique-Root invariant in Figure ?? 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*.

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

Fig. 30: Unique-Root

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

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

Fig. 31: Unique-Reachable

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

A.3 Soundness Proof of Atoms

In this section, we do proofs to show the soundness of each type rule for each atomic actions.

Lemma 13 (Unlink).

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

Proof. We assume

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

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

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 **??** as

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

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

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

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

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

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

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

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

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

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 **??** is determined by operational semantics.

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

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

?? follows from **??**

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

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 **??** and choices on maps makes $(\sigma_1', O_1', U_1', T_1', F_1')$ in denotation

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

In the rest of the proof, we prove **??**, **??** 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 **??**, we need to show that each of the memory axioms in Section **??** 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 ??** and **??** follow from framing assumption(**??**-**??**), denotations of the precondition(**??**) and **??.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 (**??**) we know that all heap locations reached by following ρ and $\rho.f_1$ are observed as iterator tid including the final reached heap locations(iterator $tid \in O_1'(\sigma'.h^*(\sigma.rt,\rho))$ and iterator $tid \in O_1'(\sigma'.h^*(\sigma.rt,\rho.f_1))$). **??** is determined directly by operational semantics.

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

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

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

from which **??** follows.

Case 5. - **OW** By **??**.**OW**, **??**, **??**.

Case 6. - **RWOW** By **??**.**RWOW**, **??** and **??**.

Case 7. - **IFL** By **??**.**WULK**, **??**.**RINFL**, **??**.**IFL**, **??** and choice of F'_1.

Case 8. - **FLR** By choice of F'_1 and **??**.

Case 9. - **WULK** By **??**, **??** and **??**.

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

Case 11. - **AWRT** By **??**.

Case 12. - **HD** By **??**.**OW**(proved), **??**.**HD** and **??**.

Case 13. - **WNR** By **??**.**WNR**, **??**, **??** and **??**.

Case 14. - **RINFL** By **??**, **??**.**RINFL**, choice of F'_1 and **??**.

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

$$\begin{aligned}\forall_{o',f'}.\, \sigma'.h(o',f') = o_y \implies &\textsf{unlinked} \in O'_1(o') \\ &\vee (\textsf{freeable} \in O'_1(o'))\end{aligned} \tag{34}$$

which follows from **??**, **??**.**OW**, operational semantics(**??**) and **??**. If o' were observed as $\textsf{iterator}$ then that would conflict with **??**.**UNQR**.

Case 16. - **UNQRT**: By **??**.**UNQRT**, **??** and **??**.

To prove **??** 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

$$\sigma_2.l \in T_2 \to (\sigma_2.h = \sigma_2'.h \land \sigma_2.l = \sigma_2'.l) \tag{35}$$
$$l \in T_2 \to F_2 = F_2' \tag{36}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \to o \in dom(\sigma_2.h) \tag{37}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \to o \in dom(\sigma_2'.h) \tag{38}$$
$$O_2 = O_2' \land U_2 = U_2' \land T_2 = T_2' \land \sigma_2.R = \sigma_2'.R \land \sigma_2.rt = \sigma_2'.rt \tag{39}$$
$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{40}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \to o \in dom(h) \tag{41}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \to o \in dom(h') \tag{42}$$

To prove all relations (**??**-**??**) we assume **??** 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.

?? and **??** follow from **??** and choice of F_2'. **??**, **??** and **??** are determined by choice of σ_2', operational semantic and choices made on maps related to the assertions.

By assuming **??** we show **??**. **??** and **??** follow trivially. **??** follows from choice of σ_2', **??** and **??**.

To prove **??** 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 **??** that

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

Both together with **??** and **??** proves **??**.

To show **??** we consider two cases: $\sigma_1'.h \cap \sigma_2'.h = \emptyset$ and $\sigma_1'.h \cap \sigma_2'.h \neq \emptyset$. First is trivial. Second follows from **??**.**OW-HD** and **??**.**OW-HD**. **??**, **??** and **??** are trivial by choices on related maps and semantics of composition operations on them. All compositions shown let us to derive conclusion for $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$.

Lemma 14 (Replace).

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

Proof. We assume

$$(\sigma, O, U, T, F)\ \in [\![\Gamma,\, p : \mathsf{rcuItr}\, \rho \mathcal{N}\, , r : \mathsf{rcuItr}\, \rho' \mathcal{N}'\, , n : \mathsf{rcuFresh}\, \mathcal{N}'']\!]_{M,tid} * \{m\} \tag{43}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{44}$$

From assumptions in the type rule of T-Replace 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 **??** 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

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

?? 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

$$O'_1 = O_1(s(n, tid))[\mathsf{fresh} \mapsto \mathsf{iterator}\, tid] \tag{70}$$

?? follows from **??**

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

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 **??** 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 **??**, **??** 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 **??**, we need to show that each of the memory axioms in Section **??** holds for the state $(\sigma', O'_1, U'_1, T'_1, F'_1)$.

Case 17. - **UNQR** Let o_p be $\sigma.s(p, tid)$, o_r be $\sigma.s(r, tid)$ and o_n be $\sigma.s(n, tid)$. **??** and **??** follow from framing assumption(**??**-**??**), denotations of the precondition(**??**), **??**.**FR** and **??**.**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 **??** and paths in **??**) are unique(From **??**.**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 (**??**) 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 **??**, **??**, **??** and **??**.**FR**.

from which we conclude **??** and **??**

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

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

from which **??** follows.

Case 18. - **OW** By **??**.**OW**, **??**, **??** and **??**.

Case 19. - **RWOW** By **??**.**RWOW**, **??**, **??** and **??**

Case 20. - **AWRT** Trivial.

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

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

Case 23. - **FPI** By **??**.

Case 24. - **WULK** Determined by operational semantics By **??**.**WULK**, **??**, **??** and operational semantics.

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

Case 26. - **HD**

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

Case 28. - **RINFL** Determined by operational semantics(**??**) and **??**.**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 **??**, **??**.**OW** and determined by operational semantics(**??**), **??**, **??**. If o' were observed as $\mathsf{iterator}$ then that would conflict with **??**.**UNQR**.

Case 30. - **UNQRT** By **??**.**UNQRT**, **??**, **??** and **??**.

To prove **??**, 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

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

To prove all relations (**??**-**??**) we assume **??** 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 **??** we show **??**. **??** and **??** follow trivially. **??** follows from choice of σ_2', **??** and **??**.

?? and **??** follow from **??** and choice of F_2'. **??**, **??** and **??** are determined by choice of σ_2', operational semantics and choices made on maps related to the assertions.

To prove **??** 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 **??** and **??**

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

From **??** we know that

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

Both together with **??** and **??** proves **??**.

For case **??**

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

From **??** we know that

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

Both together with **??** and **??** proves **??**.

To show **??** we consider two cases: $\sigma_1' \cap \sigma_2' = \emptyset$ and $\sigma_1' \cap \sigma_2' \neq \emptyset$. First is trivial. Second follows from **??**.**OW-HD** and **??**.**OW-HD**. **??**, **??** and **??** 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 15 (Insert).

$$\begin{aligned}&[\![p.f := n]\!](\lfloor[\![\Gamma,\, p : \textsf{\textit{rcultr}}\, \rho\, \mathcal{N}\,, r : \textsf{\textit{rcultr}}\, \rho_1\, \mathcal{N}_2\,, n : \textsf{\textit{rcuFresh}}\, \mathcal{N}_1]\!]_{M,tid} * \{m\}\rfloor) \subseteq \\ &\qquad \lfloor[\![\Gamma\,, p : \textsf{\textit{rcultr}}\, \rho\, \mathcal{N}([f \rightharpoonup r \setminus n])\,, n : \textsf{\textit{rcultr}}\, \rho_1\, \mathcal{N}_1\,, r : \textsf{\textit{rcultr}}\, \rho_2\, \mathcal{N}_2]\!] * \mathcal{R}(\{m\})\rfloor\end{aligned}$$

Proof. We assume

$$(\sigma, O, U, T, F)\ \in [\![\Gamma,\, p : \mathsf{rcultr}\, \rho\, \mathcal{N}\,, r : \mathsf{rcultr}\, \rho_1\, \mathcal{N}_2\,, n : \mathsf{rcuFresh}\, \mathcal{N}_1]\!]_{M,tid} * \{m\} \tag{88}$$

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{89}$$

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

$$\mathsf{FV}(\Gamma) \cap \{p, r, n\} = \emptyset \tag{90}$$

$$\rho.f = \rho_1 \text{ and } \rho.f_4 = \rho_2 \text{ and } \mathcal{N}(f) = r \tag{91}$$

$$\mathcal{N}(f) = \mathcal{N}_1(f_4) \text{ and } \forall_{f_2 \in dom(\mathcal{N}_1)}.\, f_4 \neq f_2 \implies \mathcal{N}_1(f_2) = \mathsf{null} \tag{92}$$

$$\forall_{x \in \Gamma, \mathcal{N}_3, \rho_3, f_1, y}.\, (x : \mathsf{rcultr}\, \rho_3\, \mathcal{N}_3([f_1 \rightharpoonup y])) \implies (\forall_{\rho_4 \neq \epsilon}.\, \neg\mathsf{MayAlias}(\rho_3, \rho.\rho_4)) \tag{93}$$

We split the composition in **??** as

$$(\sigma_1, O_1, U_1, T_1, F_1) \in [\![\Gamma,\ p : \mathsf{rcuItr}\,\rho\,\mathcal{N}\,,\, r : \mathsf{rcuItr}\,\rho_1\,\mathcal{N}_2\,,\, n : \mathsf{rcuFresh}\,\mathcal{N}_1]\!]_{M,tid} \tag{94}$$

$$(\sigma_2, O_2, U_2, T_2, F_2) = m \tag{95}$$

$$O_1 \bullet_O O_2 = O \tag{96}$$

$$\sigma_1 \bullet_s \sigma_2 = \sigma \tag{97}$$

$$U_1 \cup U_2 = U \tag{98}$$

$$T_1 \cup T_2 = T \tag{99}$$

$$F_1 \uplus F_2 = F \tag{100}$$

$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1) \tag{101}$$

$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{102}$$

We must show $\exists_{\sigma_1',\sigma_2',O_1',O_2',U_1',U_2',T_1',T_2',F_1',F_2'}$ such that

$$(\sigma_1', O_1', U_1', T_1', F_1') \in [\![p : \mathsf{rcuItr}\,\rho\,\mathcal{N}([f \rightharpoonup r \setminus n])\,,\, n : \mathsf{rcuItr}\,\rho_1\,\mathcal{N}_1\,,\, r : \mathsf{rcuItr}\,\rho_2\,\mathcal{N}_2\,,\, \Gamma]\!]_{M,tid} \tag{103}$$

$$(\sigma_2', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{104}$$

$$O_1' \bullet_O O_2' = O' \tag{105}$$

$$\sigma_1' \bullet_s \sigma_2' = \sigma' \tag{106}$$

$$U_1' \cup U_2' = U' \tag{107}$$

$$T_1' \cup T_2' = T' \tag{108}$$

$$F_1' \uplus F_2' = F' \tag{109}$$

$$\mathsf{WellFormed}(\sigma_1', O_1', U_1', T_1', F_1') \tag{110}$$

$$\mathsf{WellFormed}(\sigma_2', O_2', U_2', T_2', F_2') \tag{111}$$

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

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

We know that changes in observation map are

$$O_1' = O_1(s(n, tid))[\mathsf{fresh} \mapsto \mathsf{iterator}\ tid] \tag{113}$$

?? follows from **??**

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

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

$$[\![p : \mathsf{rcuItr}\,\rho\,\mathcal{N}(f \rightharpoonup r \setminus n)\,,\, n : \mathsf{rcuItr}\,\rho_1\,\mathcal{N}_1\,,\, r : \mathsf{rcuItr}\,\rho_2\,\mathcal{N}_2\,,\, \Gamma]\!]_{M,tid}$$

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

Proofs for **OW**, **RWOW**, **AWRT**, **IFL**, **WULK**, **FLR**, **FPI**, **WF**, **FR**, **HD**, **WNR**, **RINFL** and **ULKR**. The proof of **UNQR** is similar to the ones we did for Lemma **??** and Lemma **??** with a simpler fact to prove: we assume framing conditions **??**-**??** together with the **??**.**UNQR** and **??**.**FR** which makes **??UNQR** trivial.

To prove **??**, 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

$$\sigma_2.l \in T_2 \to (\sigma_2.h = \sigma'_2.h \wedge \sigma_2.l = \sigma'_2.l) \tag{115}$$
$$l \in T_2 \to F_2 = F'_2 \tag{116}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \to o \in dom(\sigma_2.h) \tag{117}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \to o \in dom(\sigma'_2.h) \tag{118}$$
$$O_2 = O'_2 \wedge U_2 = U'_2 \wedge T_2 = T'_2 \wedge \sigma_2.R = \sigma'_2.R \wedge \sigma_2.rt = \sigma'_2.rt \tag{119}$$
$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma'_2.s(x,t) \tag{120}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \to o \in dom(h) \tag{121}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \to o \in dom(h') \tag{122}$$

To prove all relations (**??**-**??**) we assume **??** 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 **??** we show **??**. **??** and **??** follow trivially. **??** follows from choice of σ'_2 and **??**.

?? and **??** follow from **??** and choice of F'_2. **??**, **??** and **??** are determined by choice of σ'_2, operational semantics and choices made on maps related to the assertions.

?? follows from assumptions **??**, **??** and choice of O'_2 as O_2.

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

Lemma 16 (ReadStack).

$$\llbracket z := x \rrbracket(\lfloor \llbracket \Gamma, z : \_, x : \textit{rcultr}\ \rho\ \mathcal{N} \rrbracket_{M,tid} * \{m\} \rfloor) \subseteq$$
$$\lfloor \llbracket \Gamma, x : \textit{rcultr}\ \rho\ \mathcal{N}, z : \textit{rcultr}\ \rho\ \mathcal{N} \rrbracket * \mathcal{R}(\{m\}) \rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F) \in \llbracket \Gamma, z : \_, x : \mathsf{rcultr}\ \rho\ \mathcal{N} \rrbracket_{M,tid} * \{m\} \tag{123}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{124}$$

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

$$\mathsf{FV}(\Gamma) \cap \{z\} = \emptyset \tag{125}$$

We split the composition in **??** as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![\Gamma\,, z : \_, x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} \tag{126}$$
$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{127}$$
$$\sigma_1 \bullet \sigma_2 = \sigma \tag{128}$$
$$O_1 \bullet_O O_2 = O \tag{129}$$
$$U_1 \cup U_2 = U \tag{130}$$
$$T_1 \cup T_2 = T \tag{131}$$
$$F_1 \uplus F_2 = F \tag{132}$$
$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1) \tag{133}$$
$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{134}$$

We must show $\exists_{\sigma_1', \sigma_2', O_1', O_2', U_1', U_2', T_1', T_2', F_1', F_2'}$ such that

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

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

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

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

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

?? follows from **??**

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

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 **??** and choices on maps makes $(\sigma_1', O_1', U_1', T_1', F_1')$ 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 **??**, **??** and show the composition of $(\sigma_1', O_1', U_1', T_1', F_1')$ and $(\sigma_2', O_2', U_2', T_2', F_2')$. **??** follows from **??** trivially.

To prove **??**, 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 (**??**-**??**) we assume **??** 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 **??** we show **??**. **??**, **??**, **??** and **??** follow trivially. **??** follows from choice of σ_2' and **??**(determined by operational semantics).

?? and **??** follow from **??** and choice of F_2'. **??**, **??** and **??** are determined by choice of σ_2', operational semantics and choices made on maps related to the assertions.

??-**??** are trivial by choices on related maps and semantics of the composition operators for these maps. **??** follows trivially from **??**. 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 17 (ReadHeap).

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

Proof. We assume

$$(\sigma, O, U, T, F) \in [\![\Gamma, z : \mathsf{rcultr}\,\_, x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} * \{m\} \tag{155}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{156}$$

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

$$\mathsf{FV}(\Gamma) \cap \{z\} = \emptyset \tag{157}$$
$$\rho.f = \rho' \tag{158}$$
$$\tag{159}$$

We split the composition in **??** as

$$(\sigma_1, O_1, U_1, T_1, F_1) \in [\![\Gamma\,, z : \mathsf{rcultr}_{\,-}, x : \mathsf{rcultr}\;\; \rho\;\; \mathcal{N}]\!]_{M,tid} \tag{160}$$

$$(\sigma_2, O_2, U_2, T_2, F_2) = m \tag{161}$$

$$\sigma_1 \bullet \sigma_2 = \sigma \tag{162}$$

$$O_1 \bullet_O O_2 = O \tag{163}$$

$$U_1 \cup U_2 = U \tag{164}$$

$$T_1 \cup T_2 = T \tag{165}$$

$$F_1 \uplus F_2 = F \tag{166}$$

$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1) \tag{167}$$

$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2) \tag{168}$$

We must show $\exists_{\sigma'_1,\sigma'_2,O'_1,O'_2,U'_1,U'_2,T'_1,T'_2,F'_1,F'_2}$ such that

$$(\sigma', O'_1, U'_1, T'_1, F_1) \in [\![\Gamma\,, x : \mathsf{rcultr}\,\rho \mathcal{N}[f \mapsto z]\,, z : \mathsf{rcultr}\,\rho'\,\mathcal{N}_\emptyset]\!]_{M,tid} \tag{169}$$

$$\mathcal{N}(f) = z \tag{170}$$

$$(\sigma', O'_2, U'_2, T'_2, F_2) \in \mathcal{R}(\{m\}) \tag{171}$$

$$\sigma'_1 \bullet \sigma'_2 = \sigma' \tag{172}$$

$$O'_1 \bullet_O O'_2 = O' \tag{173}$$

$$U'_1 \cup U'_2 = U' \tag{174}$$

$$T'_1 \cup T'_2 = T' \tag{175}$$

$$F'_1 \uplus F'_2 = F' \tag{176}$$

$$\mathsf{WellFormed}(\sigma'_1, O'_1, U'_1, T'_1, F'_1) \tag{177}$$

$$\mathsf{WellFormed}(\sigma'_2, O'_2, U'_2, T'_2, F'_2) \tag{178}$$

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

$$\sigma'_1 = \sigma_1[s(x, tid) \mapsto o_z] \tag{179}$$

?? is determined directly from operational semantics.

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

$$O'_1 = O_1 \tag{180}$$

?? follows from **??**

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{181}$$

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 **??** and choices on maps makes $(\sigma'_1, O'_1, U'_1, T'_1, F'_1)$ 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 **??**, **??** 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 **??**, we need to show that each of the memory axioms in Section **??** holds for the state $(\sigma', O'_1, U'_1, T'_1, F'_1)$.

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

Case 32. - **OW** By **??**, **??** and **??**.**OW**

Case 33. - **RWOW** By **??**, **??** and **??**.**RWOW**

Case 34. - **AWRT** Trivial.

Case 35. - **IFL** By **??**, **??**.**WULK**, **??**, choice of F'_1 and operational semantics.

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

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

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

Case 39. - **HD**

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

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

Case 42. - **ULKR** Trivial.

Case 43. - **UNQRT** By **??**.**UNQRT**, **??** and **??**.

To prove **??**, 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

$$\sigma_2.l \in T_2 \to (\sigma_2.h = \sigma'_2.h \land \sigma_2.l = \sigma'_2.l) \tag{182}$$
$$l \in T_2 \to F_2 = F'_2 \tag{183}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \to o \in dom(\sigma_2.h) \tag{184}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \to o \in dom(\sigma'_2.h) \tag{185}$$
$$O_2 = O'_2 \land U_2 = U'_2 \land T_2 = T'_2 \land \sigma_2.R_2 = \sigma'_2.R_2 \land \sigma_2.rt = \sigma'_2.rt \tag{186}$$
$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma'_2.s(x,t) \tag{187}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \to o \in dom(h) \tag{188}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \to o \in dom(h') \tag{189}$$

To prove all relations (**??**-**??**) we assume **??** 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 **??** we show **??**. **??** and **??** follows trivially. **??** follows from choice of σ'_2 and **??**(determined by operational semantics).

?? and **??** follow from **??** and choice of F'_2.**??**, **??** and **??** are determined by choice of σ'_2, operational semantics and choices made on maps related to the assertions.

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

Lemma 18 (WriteFreshField).

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

Proof. We assume

$$(\sigma, O, U, T, F)\ \in [\![\Gamma, p : \mathsf{rcuFresh}\, \mathcal{N}'_{f,\emptyset},\ x : \mathsf{rcuItr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} * \{m\} \tag{190}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{191}$$

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

$$z : \mathsf{rcuItr}\, \rho.f\ \_ \text{ and } \mathcal{N}(f) = z \text{ and } f \notin \mathit{dom}(\mathcal{N}') \tag{192}$$

We split the composition in **??** as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![\Gamma, p : \mathsf{rcuFresh}\, \mathcal{N}'_{f,\emptyset},\ x : \mathsf{rcuItr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} \tag{193}$$
$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{194}$$
$$\sigma_1 \bullet \sigma_2 = \sigma \tag{195}$$
$$O_1 \bullet_O O_2 = O \tag{196}$$
$$U_1 \cup U_2 = U \tag{197}$$
$$T_1 \cup T_2 = T \tag{198}$$
$$F_1 \uplus F_2 = F \tag{199}$$
$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1) \tag{200}$$
$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{201}$$

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

$$(\sigma', O_1', U_1', T_1', F_1') \in [\![\Gamma \,, p : \mathsf{rcuFresh}\ \ \mathcal{N}(\cup_{f \rightharpoonup z}) \,, x : \mathsf{rcuItr}\, \rho \mathcal{N}([f \rightharpoonup z])]\!]_{M,tid} \tag{202}$$

$$\mathcal{N}(f) = z \wedge \mathcal{N}'(f) = z \tag{203}$$

$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{204}$$

$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{205}$$

$$O_1' \bullet_O O_2' = O' \tag{206}$$

$$U_1' \cup U_2' = U' \tag{207}$$

$$T_1' \cup T_2' = T' \tag{208}$$

$$F_1' \uplus F_2' = F' \tag{209}$$

$$\mathsf{WellFormed}(\sigma_1', O_1', U_1', T_1', F_1') \tag{210}$$

$$\mathsf{WellFormed}(\sigma_2', O_2', U_2', T_2', F_2') \tag{211}$$

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

$$\sigma' = \sigma[h(s(p, tid), f) \mapsto s(z, tid)] \tag{212}$$

There exists no change in the observation of heap locations

$$O_1' = O_1 \tag{213}$$

?? follows from **??**

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{214}$$

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

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

In the rest of the proof, we prove **??** and **??** 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 **??**, we need to show that each of the memory axioms in Section **??** holds for the state $(\sigma', O_1', U_1', T_1', F_1')$.

Case 44. - **UNQR** By **??**.**UNQR**, **??**.**FR**(proved) and $\sigma.rt = \sigma.rt'$.

Case 45. - **OW** By **??**,**??** and **??**.**OW**

Case 46. - **RWOW** By **??**, **??** and **??**.**RWOW**

Case 47. - **AWRT** Trivial.

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

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

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

Case 51. - **WF** By **??**.**WF**, **??**, **??** and operational semantics(**??**).

Case 52. - **FR** By **??**.**FR**, **??**, **??** and operational semantics(**??**).

Case 53. - **FNR** By **??**.**FNR**, **??**, **??** and operational semantics(**??**).

Case 54. - **FPI** By **??**.**FPI**, **??** and **??**

Case 55. - **HD**

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

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

Case 58. - **ULKR** Trivial.

Case 59. - **UNQRT** By **??**.**UNQRT**, **??** and **??**.

To prove **??**, 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

$$\sigma_2.l \in T_2 \to (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{215}$$
$$l \in T_2 \to F_2 = F_2' \tag{216}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \to o \in dom(\sigma_2.h) \tag{217}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \to o \in dom(\sigma_2'.h) \tag{218}$$
$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{219}$$
$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{220}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \to o \in dom(h) \tag{221}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \to o \in dom(h') \tag{222}$$

To prove all relations (**??**-**??**) we assume **??** which is to assume T_2 as subset of reader threads and **??**. 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 **??** we show **??**. **??** and **??** follows trivially. **??** follows from choice of σ_2' and **??**(determined by operational semantics).

?? and **??** follow from **??** and choice of F_2'. **??** are determined by operational semantics, choice of σ_2' and choices made on maps related to the assertions.

??-**??** are trivial by choices on related maps and semantics of the composition operators for these maps. **??** and **??** follow from choice of σ_2'.

$O_1' \bullet O_2'$ follows from **??**, **??** and choice of O_2.

We assume $\sigma_1.h \bullet \sigma_2.h$. We know from **??** that $f \notin dom(\mathcal{N}')$. From **??**, **??**-**??**.**FNR**, **??**-**??**.**RITR** and **??**-**??**.**WNR** we show $\sigma_1'.h \bullet \sigma_2'.h$ (with choices for other maps in the machine state we show **??**). 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 (Sycn).

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

Proof. We assume

$$(\sigma, O, U, T, F) \in \llbracket \Gamma \rrbracket_{M,tid} * \{m\} \tag{223}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{224}$$

We split the composition in **??** as

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

We must show $\exists_{\sigma_1', \sigma_2', O_1', O_2', U_1', U_2', T_1', T_2'}, F_1', F_2'$ such that

$$(\sigma', O_1', U_1', T_1', F_1') \in \llbracket \Gamma[\overline{x : \mathsf{freeable}/x : \mathsf{unlinked}}] \rrbracket_{M,tid} \tag{234}$$
$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{235}$$
$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{236}$$
$$O_1' \bullet_O O_2' = O' \tag{237}$$
$$U_1' \cup U_2' = U' \tag{238}$$
$$T_1' \cup T_2' = T' \tag{239}$$
$$\tag{240}$$
$$F_1' \uplus F_2' = F' \tag{241}$$
$$\mathsf{WellFormed}(\sigma', O_1', U_1', T_1', F_1') \tag{242}$$
$$\mathsf{WellFormed}(\sigma', O_2', U_2', T_2', F_2') \tag{243}$$

We also know from operational semantics that `SyncStart` changes

$$\sigma_1'.B = \sigma_1.B[\emptyset \mapsto R] \tag{244}$$

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

$$\sigma_1' = \sigma_1 \tag{245}$$

There exists no change in the observation of heap locations

$$O'_1 = O_1(\forall_{x \in \Gamma}.\, s(x, tid))[\mathsf{unlinked} \mapsto \mathsf{freeable}] \tag{246}$$

and we pick free list to be

$$F'_1 = F_1(\forall_{x:\mathsf{unlinked} \in \Gamma, T \subseteq R}.\, s(x, tid)[T \mapsto \{\emptyset\}]) \tag{247}$$

?? follows from **??**

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{248}$$

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 **??** 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 **??** and **??** 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 **??**, we need to show that each of the memory axioms in Section **??** holds for the state $(\sigma', O'_1, U'_1, T'_1, F'_1)$ which is trivial by assuming **??**. We also know **??**(as we showed the support of state to the denotation).

To prove **??**, we need to show interference relation

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

which by definition means that we must show

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma'_2.h \land \sigma_2.l = \sigma'_2.l) \tag{249}$$
$$l \in T_2 \rightarrow F_2 = F'_2 \tag{250}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{251}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma'_2.h) \tag{252}$$
$$O_2 = O'_2 \land U_2 = U'_2 \land T_2 = T'_2 \land \sigma_2.R = \sigma'_2.R \land \sigma_2.rt = \sigma'_2.rt \tag{253}$$
$$\forall x, t \in T_2.\, \sigma_2.s(x, t) = \sigma'_2.s(x, t) \tag{254}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h) \tag{255}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h') \tag{256}$$

To prove all relations (**??**-**??**) we assume **??** which is to assume T_2 as subset of reader threads and **??**. 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 **??** we show **??**. **??** and **??** follows trivially. **??** follows from choice of σ'_2 and **??**(determined by operational semantics).

?? and **??** follow from **??**. **??** are determined by choice of σ'_2 and operational semantics and choices made on maps related to the assertions.

??-**??** follow from **??**-**??** trivially by choices on maps of logical state and semantics of composition operators. **??** follow from **??**, **??**-**??** 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 20 (Alloc).

$$\llbracket x := new \rrbracket(\lfloor \llbracket \Gamma , x : \textit{undef} \rrbracket_{M,tid} * \{m\} \rfloor) \subseteq \\ \lfloor \llbracket \Gamma , x : \textit{rcuFresh}\,\mathcal{N}_{\emptyset} \rrbracket * \mathcal{R}(\{m\}) \rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F) \in \llbracket \Gamma , x : \mathsf{undef} \rrbracket_{M,tid} * \{m\} \rfloor) \tag{257}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{258}$$

We split the composition in **??** as

$$(\sigma, O_1, U_1, T_1, F_1) \in \llbracket \Gamma , x : \mathsf{undef} \rrbracket_{M,tid} \tag{259}$$
$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{260}$$
$$\sigma_1 \bullet \sigma_2 = \sigma \tag{261}$$
$$O_1 \bullet_O O_2 = O \tag{262}$$
$$U_1 \cup U_2 = U \tag{263}$$
$$T_1 \cup T_2 = T \tag{264}$$
$$F_1 \uplus F_2 = F \tag{265}$$
$$\mathsf{WellFormed}(\sigma, O_1, U_1, T_1, F_1) \tag{266}$$
$$\mathsf{WellFormed}(\sigma, O_2, U_2, T_2, F_2) \tag{267}$$

We must show $\exists_{O'_1,O'_2,U'_1,U'_2,T'_1,T'_2,F'_1,F'_2}$ such that

$$(\sigma', O'_1, U'_1, T'_1, F'_1) \in \llbracket \Gamma , x : \mathsf{rcuFresh}\,\mathcal{N}_{\emptyset} \rrbracket \tag{268}$$
$$(\sigma', O'_2, U'_2, T'_2, F'_2) \in \mathcal{R}(\{m\}) \tag{269}$$
$$\sigma'_1 \bullet \sigma'_2 = \sigma' \tag{270}$$
$$O'_1 \bullet_O O'_2 = O' \tag{271}$$
$$U'_1 \cup U'_2 = U' \tag{272}$$
$$T'_1 \cup T'_2 = T' \tag{273}$$
$$F'_1 \uplus F'_2 = F' \tag{274}$$
$$\mathsf{WellFormed}(\sigma', O'_1, U'_1, T'_1) \tag{275}$$
$$\mathsf{WellFormed}(\sigma', O'_2, U'_2, T'_2) \tag{276}$$

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

$$\sigma' = \sigma[h(\ell) \mapsto \mathsf{nullmap}] \tag{277}$$

There exists no change in the observation of heap locations

$$O'_1 = O_1(\ell)[\mathsf{undef} \mapsto \mathsf{fresh}] \tag{278}$$

?? follows from **??**

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{279}$$

Let T'_1 to be T_1. Undefined map and free list need not change so we can pick U'_1 as U_1 and F'_1 as F_1 and show(**??**) 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 **??**, **??** and $(\sigma'_1, O'_1, U'_1, T'_1, F'_1)$ and $(\sigma'_2, O'_2, U'_2, T'_2, F'_2)$. To prove **??**, we need to show that each of the memory axioms in Section **??** holds for the state $(\sigma', O'_1, U'_1, T'_1, F'_1)$.

Case 60. - **UNQR** Determined by **??** and operational semantics(ℓ is fresh-unique).

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

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 **??**.

Case 65. - **WF** By **??**, **??** and **??**.

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

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

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

Case 69. - **HD**

Case 70. - **WNR** By **??**.

To prove **??**, 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

$$\sigma_2.l \in T_2 \to (\sigma_2.h = \sigma'_2.h \wedge \sigma_2.l = \sigma'_2.l) \tag{280}$$
$$l \in T_2 \to F_2 = F'_2 \tag{281}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \to o \in dom(\sigma_2.h) \tag{282}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \to o \in dom(\sigma'_2.h) \tag{283}$$
$$O_2 = O'_2 \wedge U_2 = U'_2 \wedge T_2 = T'_2 \wedge \sigma_2.R = \sigma'_2.R \wedge \sigma_2.rt = \sigma'_2.rt \tag{284}$$
$$\forall x, t \in T.\, \sigma_2.s(x,t) = \sigma'_2.s(x,t) \tag{285}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \to o \in dom(h) \tag{286}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \to o \in dom(h') \tag{287}$$

To prove all relations (**??**-**??**) we assume **??** which is to assume T_2 as subset of reader threads and **??**. 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 **??** and choices on maps we show **??**. **??** and **??** follow trivially. **??** follows from choice of σ_2' and **??**(determined by operational semantics). **??**-**??** follow from **??**-**??**, semantics of compositions operators and choices made for maps of the logical state.

?? and **??** follow from **??** and choice on F_2'. **??** 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, **??** follows from **??** 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 (Free).

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

Proof. We assume

$$(\sigma, O, U, T, F) \in [\![x : \mathsf{freeable}]\!]_{M,tid} * \{m\} \rfloor) \tag{288}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{289}$$

We split the composition in **??** as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![x : \mathsf{freeable}]\!]_{M,tid} \tag{290}$$
$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{291}$$
$$\sigma_1 \bullet \sigma_2 = \sigma \tag{292}$$
$$O_1 \bullet_O O_2 = O \tag{293}$$
$$U_1 \cup U_2 = U \tag{294}$$
$$T_1 \cup T_2 = T \tag{295}$$
$$F_1 \uplus F_2 = F \tag{296}$$
$$\mathsf{WellFormed}(\sigma, O_1, U_1, T_1, F_1) \tag{297}$$
$$\mathsf{WellFormed}(\sigma, O_2, U_2, T_2, F_2) \tag{298}$$

We must show $\exists_{O'_1,O'_2,U'_1,U'_2,T'_1,T'_2,F'_1,F'_2}$ such that

$$(\sigma', O'_1, U'_1, T'_1, F'_1) \in [\![x : \mathsf{undef}]\!] \tag{299}$$
$$(\sigma', O'_2, U'_2, T'_2, F'_2) \in \mathcal{R}(\{m\}) \tag{300}$$
$$\sigma'_1 \bullet \sigma'_2 = \sigma' \tag{301}$$
$$O'_1 \bullet_O O'_2 = O' \tag{302}$$
$$U'_1 \cup U'_2 = U' \tag{303}$$
$$T'_1 \cup T'_2 = T' \tag{304}$$
$$F'_1 \uplus F'_2 = F' \tag{305}$$
$$\mathsf{WellFormed}(\sigma', O'_1, U'_1, T'_1, F'_1) \tag{306}$$
$$\mathsf{WellFormed}(\sigma', O'_2, U'_2, T'_2, F'_2) \tag{307}$$

From operational semantics we know that

$$\forall_{f,o'}.\, rt \neq s(x,tid) \wedge o' \neq s(x,tid) \implies h(o',f) = h'(o',f) \wedge \forall_f.\, h'(o,f) = \mathsf{undef} \tag{308}$$

$$O'_1 = O_1(s(x,tid))[\mathsf{freeable} \mapsto \mathsf{undef}] \tag{309}$$

$$F'_1 = F_1 \setminus \{s(x,tid) \mapsto \{\emptyset\}\} \tag{310}$$

$$U'_1 = U_1 \cup \{(x,tid)\} \tag{311}$$

?? follows from **??**

$$T_1 = \{tid\} \text{ and } tid = \sigma.l \tag{312}$$

Let T'_1 to be T_1. All **??**-**??** show(**??**) 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 **??**, **??** 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 **??**, we need to show that each of the memory axioms in Section **??** holds for the state $(\sigma', O'_1, U'_1, T'_1, F'_1)$ and it it trivial by **??**-**??** and **??**.

To prove **??**, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{313}$$
$$l \in T_2 \rightarrow F_2 = F_2' \tag{314}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{315}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{316}$$
$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{317}$$
$$\forall x, t \in T.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{318}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h) \tag{319}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \rightarrow o \in dom(h') \tag{320}$$

To prove all relations (**??**-**??**) we assume **??** which is to assume T_2 as subset of reader threads and **??**. 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 **??** and choices on maps we show **??**. **??** and **??** follow trivially. **??** follows from choice of σ_2' and **??**(determined by operational semantics). **??**-**??** follow from **??**-**??**, semantics of composition operators and choices on related maps.

?? and **??** follow from **??** and choice on F_2'. **??** 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 **??** straightforward. Since other machine components do not change(determined by operational semantics), **??** follows from **??**, **??** 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 (RReadStack).

$$\llbracket z := x \rrbracket (\lfloor \llbracket \Gamma, z : \textit{rcultr}, x : \textit{rcultr} \rrbracket_{R,tid} * \{m\} \rfloor) \subseteq \\ \lfloor \llbracket \Gamma, x : \textit{rcultr}, z : \textit{rcultr} \rrbracket * \mathcal{R}(\{m\}) \rfloor$$

Proof. We assume

$$(\sigma, O, U, T, F) \in \llbracket \Gamma, \Gamma, z : \mathsf{rcultr}, x : \mathsf{rcultr} \rrbracket_{R,tid} * \{m\} \tag{321}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{322}$$

We split the composition in **??** as

$$(\sigma_1, O_1, U_1, T_1, F_1) \in [\![\Gamma\, , \Gamma\, , z : \mathsf{rcultr}\, , x : \mathsf{rcultr}]\!]_{R,tid} \tag{323}$$
$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{324}$$
$$O_1 \bullet_O O_2 = O \tag{325}$$
$$\sigma_1 \bullet \sigma_2 = \sigma \tag{326}$$
$$U_1 \cup U_2 = U \tag{327}$$
$$T_1 \cup T_2 = T \tag{328}$$
$$F_1 \uplus F_2 = F \tag{329}$$
$$\mathsf{WellFormed}(\sigma, O_1, U_1, T_1, F_1) \tag{330}$$
$$\mathsf{WellFormed}(\sigma, O_2, U_2, T_2, F_2) \tag{331}$$

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

$$(\sigma', O_1', U_1', T_1', F_1') \in [\![\Gamma\, , x : \mathsf{rcultr}\, , z : \mathsf{rcultr}]\!]_{R,tid} \tag{332}$$
$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{333}$$
$$O_1' \bullet_O O_2' = O' \tag{334}$$
$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{335}$$
$$U_1' \cup U_2' = U' \tag{336}$$
$$T_1' \cup T_2' = T' \tag{337}$$
$$F_1' \uplus F_2' = F' \tag{338}$$
$$\mathsf{WellFormed}(\sigma', O_1', U_1', T_1', F_1') \tag{339}$$
$$\mathsf{WellFormed}(\sigma', O_2', U_2', T_2', F_2') \tag{340}$$

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

$$\sigma_1' = \sigma_1 \tag{341}$$

There exists no change in the observation of heap locations

$$O_1' = O_1 \tag{342}$$

?? follows from **??**

$$T_1 \subseteq R \tag{343}$$

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 **??** 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 **??**, **????**, **??** 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 **??**, we need to show that each of the memory axioms in Section **??** holds for the state $(\sigma', O'_1, U'_1, T'_1, F'_1)$ which is trivial by assuming **??** and knowing **??**, **??** and components of the state determined by operational semantics.

To prove **??**, we need to show that $\mathsf{WellFormed}$ness 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

$$\sigma_2.l \in T_2 \to (\sigma_2.h = \sigma'_2.h \land \sigma_2.l = \sigma'_2.l) \tag{344}$$
$$l \in T_2 \to F_2 = F'_2 \tag{345}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \to o \in dom(\sigma_2.h) \tag{346}$$
$$\forall tid, o.\, \mathsf{iterator}\, tid \in O_2(o) \to o \in dom(\sigma'_2.h) \tag{347}$$
$$O_2 = O'_2 \land U_2 = U'_2 \land T_2 = T'_2 \land \sigma_2.B = \sigma'_2.B \land \sigma_2.rt = \sigma'_2.rt \tag{348}$$
$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma'_2.s(x,t) \tag{349}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \to o \in dom(h) \tag{350}$$
$$\forall tid, o.\, \mathsf{root}\, tid \in O(o) \to o \in dom(h') \tag{351}$$

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 **??**-**??** trivial and **??**-**??** follow from assumptions **??**-**??**, choices made for related maps and semantics of composition operations. All compositions shown let us derive conclusion for $(\sigma'_1, O'_1, U'_1, T'_1, F'_1) \bullet (\sigma'_2, O'_2, U'_2, T'_2, F'_2)$.

Lemma 23 (RReadHeap).

$$\begin{aligned}\llbracket z := x.f \rrbracket(\lfloor \llbracket \Gamma, z : \mathit{rcultr}, x : \mathit{rcultr} \rrbracket_{R,tid} * \{m\} \rfloor) \subseteq \\ \lfloor \llbracket \Gamma, x : \mathit{rcultr}, z : \mathit{rcultr} \rrbracket * \mathcal{R}(\{m\}) \rfloor\end{aligned}$$

Proof. We assume

$$(\sigma, O, U, T, F) \in \llbracket \Gamma, z : \mathsf{rcultr}, x : \mathsf{rcultr} \rrbracket_{R,tid} * \{m\} \rfloor) \tag{352}$$
$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{353}$$

We split the composition in **??** as

$$(\sigma_1, O_1, U_1, T_1, F_1) \in \llbracket \Gamma, z : \mathsf{rcultr}, x : \mathsf{rcultr} \rrbracket_{R,tid} \tag{354}$$
$$(\sigma_2, O_2, U_2, T_2, F_2) = m \tag{355}$$
$$\sigma_1 \bullet \sigma_2 = \sigma \tag{356}$$
$$O_1 \bullet_O O_2 = O \tag{357}$$
$$U_1 \cup U_2 = U \tag{358}$$
$$T_1 \cup T_2 = T \tag{359}$$
$$F_1 \uplus F_2 = F \tag{360}$$
$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1) \tag{361}$$
$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{362}$$

We must show $\exists_{\sigma_1',\sigma_2',O_1',O_2',U_1',U_2',T_1',T_2',F_1',F_2'}$ such that

$$(\sigma_1', O_1', U_1', T_1', F_1') \in \lfloor\llbracket \Gamma , x : \mathsf{rcultr} , z : \mathsf{rcultr} \rrbracket \tag{363}$$

$$(\sigma_2', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{364}$$

$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{365}$$

$$O_1' \bullet_O O_2' = O' \tag{366}$$

$$U_1' \cup U_2' = U' \tag{367}$$

$$T_1' \cup T_2' = T' \tag{368}$$

$$\mathsf{WellFormed}(\sigma_1', O_1', U_1', T_1', F_1') \tag{369}$$

$$\mathsf{WellFormed}(\sigma_2', O_2', U_2', T_2', F_2') \tag{370}$$

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

$$\sigma_1' = \sigma_1[s(z, tid) \mapsto o_x] \tag{371}$$

There exists no change in the observation of heap locations

$$O_1' = O_1 \tag{372}$$

?? follows from **??**

$$T_1 \subseteq R \tag{373}$$

Proof is similar to Lemma **??**.

A.4 Soundness Proof of Structural Program Actions

In this section, we introduce soundness Theorem **??** 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 1 (Type System Soundness).

$$\forall_{\Gamma,\Gamma',C}.\, \Gamma \vdash C \dashv \Gamma' \implies \llbracket \Gamma \vdash C \dashv \Gamma' \rrbracket$$

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

$$\Gamma' \vdash_M C \dashv \Gamma'' \tag{374}$$

$$\Gamma \prec: \Gamma' \tag{375}$$

$$\Gamma'' \prec: \Gamma''' \tag{376}$$

$$\{\llbracket \Gamma' \rrbracket_{M,tid}\} C \{\llbracket \Gamma'' \rrbracket_{M,tid}\} \tag{377}$$

We need to show

$$\{[\![\Gamma]\!]_{M,tid}\}C\{[\![\Gamma''']\!]_{M,tid}\} \tag{378}$$

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

From **??** and Lemma **??** we know

$$[\![\Gamma]\!]_{M,tid} \sqsubseteq [\![\Gamma']\!]_{M,tid} \tag{379}$$

From **??** and **??** we know

$$[\![\Gamma'']\!]_{M,tid} \sqsubseteq [\![\Gamma''']\!]_{M,tid} \tag{380}$$

By using **??**, **??** and **??** as antecedentes of Views Logic's consequence rule, we conclude **??**.

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

$$\{[\![\Gamma]\!]_{M,tid}\} \vdash_M C_1; C_2 \dashv \{[\![\Gamma'']\!]_{M,tid}\} \tag{381}$$

We know

$$\Gamma \vdash_M C_1 \dashv \Gamma' \tag{382}$$

$$\Gamma' \vdash_M C_2 \dashv \Gamma'' \tag{383}$$

$$\{[\![\Gamma]\!]_{M,tid}\}C_1\{[\![\Gamma']\!]_{M,tid}\} \tag{384}$$

$$\{[\![\Gamma']\!]_{M,tid}\}C_2\{[\![\Gamma'']\!]_{M,tid}\} \tag{385}$$

By using **??** and **??** as the antecedents for the Views sequencing rule, we can derive the conclusion for **??**.

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

$$\Gamma \vdash_M C \dashv \Gamma \tag{386}$$

$$\Gamma(x) = \mathsf{bool} \tag{387}$$

$$\{[\![\Gamma]\!]_{M,tid}\}C\{[\![\Gamma]\!]_{M,tid}\} \tag{388}$$

Our goal is to prove

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

We prove **??** by from the consequence rule, based on the proofs of the following **??** and **??**

$$\{[\![\Gamma]\!]_{M,tid}\}\,(\mathit{assume}\,(x)\,; C)^*\,\{[\![\Gamma]\!]_{M,tid}\} \tag{390}$$

$$\{[\![\Gamma]\!]_{M,tid}\}assume\,(\neg x)\,\{[\![\Gamma]\!]_{M,tid}\} \tag{391}$$

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

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

as antecedent. We can use this antecedent together with the antecedent we know from **??**

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

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 **??** 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 $if\ (e)\ then\{C_1\}else\{C_2\}$.

$$\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 \rightharpoonup z]) \vdash_M C_1 \dashv \Gamma' \tag{392}$$
$$\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_2 \rightharpoonup z]) \vdash_M C_2 \dashv \Gamma' \tag{393}$$
$$\{[\![\Gamma,\, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 \rightharpoonup z])]\!]_{M,tid}\}C_1\{[\![\Gamma']\!]_{M,tid}\} \tag{394}$$
$$\{[\![\Gamma,\, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_2 \rightharpoonup z])]\!]_{M,tid}\}C_2\{[\![\Gamma']\!]_{M,tid}\} \tag{395}$$

Our goal is to prove

$$\begin{array}{l}\{[\![\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1|f_2 \rightharpoonup z])]\!]_{M,tid}\}\\ y = x.f_1; (assume\,(z = y)\,;C_1) + (assume\,(y \neq z)\,;C_2)\\ \{[\![\Gamma']\!]_{M,tid}\}\end{array} \tag{396}$$

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

$$\begin{array}{l}\{[\![\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1|f_2 \rightharpoonup z])]\!]_{M,tid}\}\\ (assume\,(z = y)\,;C_1) + (assume\,(y \neq z)\,;C_2)\\ \{[\![\Gamma']\!]_{M,tid}\}\end{array} \tag{397}$$

and

$$\begin{array}{l}\{[\![\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}[f_1 | f_2 \rightharpoonup z]]\!]_{M,tid}\} \\ y = x.f_1 \\ \{[\![\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 | f_2 \rightharpoonup z])[\![_{M,tid} \cap [\![x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 \rightharpoonup y])]\!]_{M,tid}\}\end{array} \tag{398}$$

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

We prove **??** from the branch rule of Views Logic based on the proofs of the following **??** and **??**

$$\begin{array}{l}\{[\![\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 | f_2 \rightharpoonup z])]\!]_{M,tid} \cap \\ [\![x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 \rightharpoonup y])]\!]_{M,tid}\} \\ (assume\,(z = y)\,; C_1) \\ \{[\![\Gamma']\!]_{M,tid}\}\end{array} \tag{399}$$

and

$$\begin{array}{l}\{[\![\Gamma, x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 | f_2 \rightharpoonup z])]\!]_{M,tid} \cap \\ [\![x : \mathsf{rcultr}\ \rho\, \mathcal{N}([f_1 \rightharpoonup y])]\!]_{M,tid}\} \\ (assume\,(z \neq y)\,; C_2) \\ [\![\Gamma']\!]_{M,tid}\}\end{array} \tag{400}$$

We show **??** 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 **??** 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

$$\Gamma_1 \vdash C_1 \dashv \Gamma_1' \tag{401}$$
$$\Gamma_2 \vdash C_2 \dashv \Gamma_2' \tag{402}$$
$$\{[\![\Gamma_1]\!]\} C_1 \{[\![\Gamma_1']\!]\} \tag{403}$$
$$\{[\![\Gamma_2]\!]\} C_2 \{[\![\Gamma_2']\!]\} \tag{404}$$

We need to show

$$\{[\![\Gamma_1, \Gamma_2]\!]\} C_1 || C_2 \{[\![\Gamma_1', \Gamma_2']\!]\} \tag{405}$$

By using **??** and **??** as antecedents to Views Logic's parallel rule, we can draw conclusion for **??**

$$\{[\![\Gamma_1]\!] * [\![\Gamma_2]\!]\} C_1 || C_2 \{[\![\Gamma_1']\!] * [\![\Gamma_2']\!]\} \tag{406}$$

Showing **??** requires showing

$$[\![\Gamma_1, \Gamma_2]\!] \sqsubseteq [\![\Gamma_1]\!] * [\![\Gamma_2]\!] \tag{407}$$

$$[\![\Gamma_1']\!] * [\![\Gamma_2']\!] \sqsubseteq [\![\Gamma_1', \Gamma_2']\!] \tag{408}$$

By using **??** and **??**(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 **??**.

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

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

We assume from the rule ToRCUWrite

$$\Gamma, y : \mathsf{rcultr}\ \_ \vdash_M C \dashv \Gamma' \tag{409}$$
$$\mathsf{FType}(f) = \mathsf{RCU} \tag{410}$$
$$\mathsf{NoFresh}(\Gamma') \tag{411}$$
$$\mathsf{NoFreeable}(\Gamma') \tag{412}$$
$$\mathsf{NoUnlinked}(\Gamma') \tag{413}$$
$$\{[\![\Gamma, y : \mathsf{rcultr}\ \_]\!]_{M,tid}\} C \{[\![\Gamma']\!]_{M,tid}\} \tag{414}$$

Our goal is to prove

$$\{[\![\Gamma]\!]_{M,tid}\} \mathsf{WriteBegin}; C; \mathsf{WriteEnd} \{[\![\Gamma']\!]_{M,tid}\} \tag{415}$$

Any case of C does not change the state(no heap update) by assumptions **??**-**??** therefore **??** follows from assumptions **??**-**??** trivially.

Lemma 24 (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 **??**.

Lemma 25 (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 **??**.

Lemma 26 (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 27 (Singleton-SubTyping-R).

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

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

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

B RCU BST Delete

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

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

```
// CASE 2: If current's right child has no left child, then current's right child
 // replaces current in the tree
$else\ if\ (current.Left == null)$
{
 $\left\{parent : rcuItr\ (Left|Right)^k\ \{(Left|Right) \mapsto current\}\right\}$
 $\left\{current : rcuItr\ (Left|Right)^k.(Left|Right)\ \{Left \mapsto null, Right \mapsto lmParent\}\right\}$
 $\left\{currentL : rcuItr\ (Left|Right)^k.(Left|Right).Left\ \{\}\right\}$
 $\left\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right\ \{\}\right\}$
 $if\ (parent.Left == current)$
   $\left\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto current\}\right\}$
   $\left\{current : rcuItr\ (Left|Right)^k.Left\ \{Left \mapsto null, Right \mapsto lmParent\}\right\}$
   $\left\{lmParent : rcuItr\ (Left|Right)^k.Left.Right\ \{\}\right\}$
   // parent.Value is greater than current.Value
   // so make current's right child a left child of parent
   $parent.Left = lmParent;$
   $\left\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto lmParent\}\right\}$
   $\{current : unlinked\}$
   $\left\{lmParent : rcuItr\ (Left|Right)^k.Left\ \{\}\right\}$
 $else$
   $\left\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto current\}\right\}$
   $\left\{current : rcuItr\ (Left|Right)^k.Right\ \{Left \mapsto null, Right \mapsto lmParent\}\right\}$
   $\left\{lmParent : rcuItr\ (Left|Right)^k.Right.Right\ \{\}\right\}$
   // parent.Value is less than current.Value
   // so make current's right child a right child of parent
   $parent.Right = lmParent;$
   $\left\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto lmParent\}\right\}$
   $\left\{lmParent : rcuItr\ (Left|Right)^k.Right\ \{\}\right\}$
   $\{current : unlinked\}$
   $SyncStart;$
   $SyncStop;$
   $\{current : freeable\}$
   $Free(current);$
   $\{current : undef\}$
}
```

```
// CASE 3: If current's right child has a left child, replace current with current's
// right child's left-most descendent
$else$
{
  $\{parent : rcuItr\ (Left|Right)^k\ \{(Left|Right) \mapsto current\}\}$
  $\{current : rcuItr\ (Left|Right)^k.(Left|Right)\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$
  $\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right\ \{\}\}$
  $\{currentL : rcuItr\ (Left|Right)^k.(Left|Right).Left\ \{\}\}$
  // We first need to find the right node's left-most child
  $BinaryTreeNode\ currentF = new;$
  $\{currentF : rcuFresh\}$
  $currentF.Right = lmParent;$
  $\{currentF : rcuFresh\ \{Right \mapsto lmParent\}\}$
  $currentF.Left = currentL;$
  $\{currentF : rcuFresh\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$
  $BinaryTreeNode\ leftmost = lmParent.Left;$
  $\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right\ \{Left \mapsto leftmost\}\}$
  $\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left\ \{\}\}$
  $if\ (lmParent.Left == null)\{$
  $\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right\ \{Left \mapsto null\}\}$
    $currentF.data = lmParent.data;$
    $if\ (parent.Left == current)\{$
      $\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto current\}\}$
      $\{current : rcuItr\ (Left|Right)^k.Left\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$
      $\{currentF : rcuFresh\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$
      //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
      $parent.Right = currentF;$
      $\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto currentF\}\}$
      $\{current : unlinked\}$
      $\{currentF : rcuItr\ (Left|Right)^k.Right\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$
      $SyncStart;$
      $SyncStop;$
      $\{current : freeable\}$
      $Free(current);$
      $\{current : undef\}$
    }
  }
```

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

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

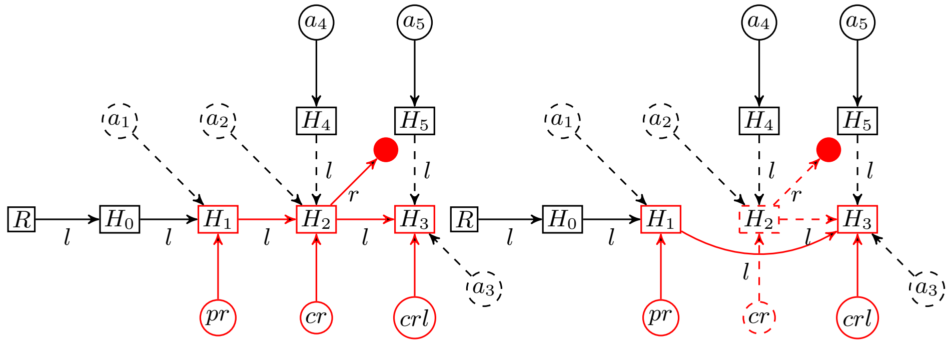
C RCU Bag with Linked-List

```
$BagNode\ head;$
$int\ member\ (int\ toRead)\ \{$
  $ReadBegin;$
  $int\ result = 0;$
  $\{parent : undef,\ head : rcuRoot\}$
  $BagNode\ parent = head;$
  $\{parent : rcuItr\}$
  $\{current : \_\}$
  $current = parent.Next;$
  $\{current : rcuItr,\ parent : rcuItr\}$
  $\{current : rcuItr\}$
  $while(current.data\ != \ toRead\&\&current.Next \neq null)\{$
    $\{parent : rcuItr\}$
    $\{current : rcuItr\}$
    $parent = current;$
    $current = parent.Next;$
    $\{parent : rcuItr\}$
    $\{current : rcuItr\}$
  $\}$
  $\{parent : rcuItr\}$
  $\{current : rcuItr\}$
  $result = current.data;$
  $ReadEnd;$
  $return\ result;$
$\}$
```

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

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

D Safe Unlinking



(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. 32: Safe unlinking of a heap node from a BST

Preserving invariants of a data structure against possible mutations under RCU semantics is challenging. Unlinking a heap node is one way of mutating the heap. To understand the importance of the locality on the possible effects of the mutation, we illustrate a setting for unlinking a heap in Figures **??** and **??**. 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 depicts `null`, l field which depicts $Left$ and r depicts $Right$ field.

The type rule for unlinking must assert the ”proper linkage” in between the heap nodes involved in the action of unlinking. We see the proper linkage relation between in Figure **??** 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, $\mathcal{N}_1([f_2 \rightharpoonup r])$ $(Left \mapsto currentL)$ 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 aliases to the resources under mutation, e.g. aliasing to `parent`, `current` and `currentL`, needs to be handled carefully. Aliasing can occur via either through object fields – via field mappings – or stack pointers – via path components. We see path aliases, a_1, a_2 and a_3, illustrated with dashed nodes and arrows to the heap nodes in Figures **??** and **??**. 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

$$(\neg\mathsf{MayAlias}(\rho_3, \{\rho, \rho_1, \rho_2\}))$$

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

Any heap node reached from root by following a path(ρ_3) deeper than the path reaching to the last heap node(*crl*) in the footprint cannot be pointed by any of the heap nodes(*pr*, *cr* and *crl*) in the footprint. We require this restriction to prevent inconsistency on path components of references, ρ_3, referring to heap nodes deeper than memory footprint

$$(\forall_{\rho_4 \neq \epsilon}. \neg\mathsf{MayAlias}(\rho_3, \rho_2.\rho_4))$$

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

One another safety achieved with framing shows up in a setting where `current` and a_2 are aliases. In the Figure **??**, both `current` and a_2 are in the rcuItr type and point to H_2. After the unlinking occurs, 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 occurs, both `currentL` and a_3 have the same path components. After the unlinking, the path of `currentL`-*crl* gets shortened as the path to heap node it points, H_3, changes to $(Left)^k.Left$. However, the path component of a_3 would not change so the path component of a_3 in the rcuItr would become inconsistent with the actual path reaching to H_3.

In addition to *path-aliasing*, there can also be aliasing via *field-mappings* which we call *field-aliasing*. We see field aliasing examples in Figures **??** and **??**: *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 **??** and **??**.

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 **??**.

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

$$\forall_{f \in dom(\mathcal{N}_1)}.\, f \neq f_2 \implies (\mathcal{N}_1(f) = \mathsf{null})$$

E Types Rules for RCU Read Section

$$\boxed{\Gamma \vdash_R \alpha \dashv \Gamma'} \quad \text{(T-ReadS)}\ \frac{z \notin \mathsf{FV}(\Gamma)}{\Gamma\,, z : \_\,, x : \mathsf{rcultr} \vdash z = x \dashv x : \mathsf{rcultr}\,, z : \mathsf{rcultr}\,, \Gamma}$$

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

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

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

$$\boxed{\Gamma \vdash_{M,R} C \dashv \Gamma'} \quad \text{(T-Branch2)}\ \frac{\Gamma(x) = \mathsf{bool} \qquad \Gamma \vdash C_1 \dashv \Gamma' \qquad \Gamma \vdash C_2 \dashv \Gamma'}{\Gamma \vdash \mathsf{if}(x)\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \dashv \Gamma'}$$

$$\text{(T-Seq)}\ \frac{\Gamma_1 \vdash C_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash C_2 \dashv \Gamma_3}{\Gamma_1 \vdash C_1\ ;\ C_2 \dashv \Gamma_3} \qquad \text{(T-Par)}\ \frac{\Gamma_1 \vdash_R C_1 \dashv \Gamma_1' \qquad \Gamma_2 \vdash_{M,R} C_2 \dashv \Gamma_2'}{\Gamma_1,\ \Gamma_2 \vdash C_1 || C_2 \dashv \Gamma_1'\,, \Gamma_2'}$$

$$\text{(T-Exchange)}\ \frac{\Gamma,\, y : T',\, x : T,\, \Gamma' \vdash C \dashv \Gamma''}{\Gamma,\, x : T,\, y : T',\, \Gamma' \vdash C \dashv \Gamma''} \qquad \text{(T-Conseq)}\ \frac{\Gamma \prec: \Gamma' \qquad \Gamma' \vdash C \dashv \Gamma'' \qquad \Gamma'' \prec: \Gamma'''}{\Gamma \vdash C \dashv \Gamma'''}$$

$$\text{(T-Skip)}\ \frac{}{\Gamma \vdash \mathsf{skip} \dashv \Gamma}$$

Fig. 33: Type Rules for Read critical section for RCU Programming