Modal Abstractions for Operating System Kernels

A Thesis

Submitted to the Faculty

of

Drexel University

by

Ismail Kuru

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

March 2025

DEDICATIONS

...

ACKNOWLEDGMENTS

...

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Modal Abstractions for Operating System Kernels
Ismail Kuru
Advisor: Dr. Colin S. Gordon

Operating-System kernels are important pieces of the software world, interacting with the hardware, and their reliability is essential, as the rest of the software world depends on it. There are certain aspects that make verifying kernels challenging, and this thesis captures principles dealing with three of these aspects.

The first challenge stems from the sharing of a resource among different kernel abstractions (and their particular instances) through an indirection mechanism employed between address (alias) and the data (virtualization). As the first contribution of this thesis, we introduce reasoning principles for understanding location virtualization in kernels. Location virtualization appears in multiple components of the kernel, including memory and file resource *virtualization*. In this thesis, we take virtual memory management (VMM) as our experimental setting to apply our reasoning principles. VMM code is a critical piece of general-purpose OS kernels, but verification of this functionality is challenging due to the complexity of the hardware interface (the page tables are updated via writes to those memory locations, using addresses that are themselves virtualized). Prior work on verification of VMM code has either only handled a single address space, trusted significant pieces of assembly code, or resorted to direct reasoning over machine semantics rather than exposing a clean logical interface. In this thesis, we introduce a modal abstraction to describe the truth of assertions relative to a specific virtual address space: $[r]$P indicating that P holds in the virtual address space rooted at r. Such modal assertions allow different address spaces to refer to each other, enabling complete verification of instruction sequences and manipulating multiple address spaces. Using them effectively requires working with other assertions, such as points-to assertions in our separation logic, relative to a given address space. We therefore define virtual points-to relations, which mimic hardware address translation, relative to a page table root.

The second challenge appears when we extend the kernel functionality. It is common for any software to expect certain API usage to follow a protocol, and it is particularly common in OS kernels, where the protocols are essentially the contracts for extension points. Any kernel, in one way or another, realizes mechanisms that allow us to plug in different implementations of the same functionality. For example, Virtual File System (VFS) enables different filesystem implementations to

coexist, or device drivers require extensions with respect to the changing device capabilities. In doing so, these mechanisms impose certain protocols which may internally have simpler or more complex usage protocols that are *compatible* with the official interface. This constitutes the second challenge, as any change in the protocol brings into question whether the validity (or reliability) of the client code is still valid or not (evolution). As the second contribution of this thesis, we want to let proofs capture and exploit the compatibility of the client proofs against the evolution of the specifications. To do so, we introduce a single-form logical abstraction for specifying protocols that are abstracted as state transition systems (STS)es. STSes are an increasingly popular means of specifying and verifying fine-grained concurrent programs. Unlike more traditional rely-guarantee-based approaches, they allow interference to be conveniently treated as a resource, transferred between threads, or even stored in other resources. However, existing STS systems leave the traditional Hoare-style rule of consequence weaker than before. Code involving STSes is verified against one particular STS, making it unusable with other similar transition systems, even when one is contained in the other. We extend the notion of entailment for STS-based logics to incorporate a form of bisimulation (as on Kripke structures) between STS systems into an extended rule of consequence. We show that a specification of a file write operation preserves its validity against a file resource usage protocol changed to handle distributed write requests.

The third challenge this thesis is concerned with is *ergonomics* of assertions that may be useful for the correct usage of the constructs that are exposed as a kernel API. One of the common constructs appearing in kernel APIs is concurrency constructs (e.g., reader-writer locks). As low-level systems, kernels are expected to be more efficient. Thus, in them, more highly optimized sharing mechanisms and data structures are implemented using these sharing mechanisms. As a critical piece, memory management in lock-free data structures is promising but is 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. In other words, the reasoning principles for this kind of semantics must refer to the representation of the *contingent-truth* referring to *the consistency of the program state with respect to a certain grace period*. 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 that 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.

In conclusion, we explain a general verification approach shaped around the concept of resource context that helps to design new modalities to verify the system code. We justify our perspective by identifying existing systems that have used modalities for systems verification successfully, arguing that they fit into the verification design pattern we articulate, and explaining how this approach might apply to other systems verification challenges.

Part I

Introduction

CHAPTER 1

INTRODUCTION

The fundamental contribution of this thesis falls into the broader context of *software reliability*. Any methodology in this context aims to build trust to the software, and program verification is one of them. What makes program verification promising is the level of assurance it provides on the subject software: once the program verification is completed on the subject software, there is a mathematical explanation, i.e. *proof*, showing why the software behaves as expected, which implies the potential of eliminating whole classes of bugs. In other words, program verification as an action of increasing software reliability aims to obtain mathematical proof showing whether the high-level specification describing the intended behavior of a program is met by the program.

Specification A verification system must allow for a language of assertions to enable specifying program behavior. Verification systems either allow programs to be annotated with the relevant assertions (e.g., loop invariants, specifications of functions in the form of pre/post conditions) [35,38], or keep the specification separate from the program source as a part of an abstract logic [16,53,54,91,100,149,153,156,157].

Proof A verification system must also provide means to obtain the witness, i.e. *the proof*, showing the fact that the specification is satisfied by the program. Obtaining a mathematical proof effort may differ according to the methodology. Well-known realizations can either fall into *truth-searching* techniques [12,15,35,38,44], or *truth-constructing* techniques [16,53,54,91,100,149,153,156,157]. Searching for proof requires a high-level specification (e.g. pre- post-conditions, and loop invariants) that is encoded

into SMT formulas (SAT formulas modulo some additional theories for real numbers, integers, or data structures such as lists, bits, and arrays). Then, the proof search is done on the optimized data structures that hold the encoded truth. Constructing a proof, which we follow as a methodology for the proofs in this thesis, requires building a set of reasoning principles, i.e. a program logic [16,53,54,91,100,149,153,156,157]. A program logic has proof rules for each of the program actions, for example, a proof rule dictating how a memory location can be accessed. The proof of the program is constructed by applying the relevant proof rule to each of the program actions. In doing so, for the sake of increasing the reliability of the proof itself, machine assistance using theorem provers [26,43,131] can be taken. All proofs except those in Part VII are constructed using a theorem prover named Roqc. The manual effort brings the advantage of constructing advanced program logics [16,53,54,91,100,149,153,156,157], and of reasoning over more advanced programs. To speak more concretely, for example, introducing reasoning principles for concurrent semantics or a higher-order reasoning for the programs requires more abstract and custom-tailored logical principles, e.g., a consistent view of concurrent threads on the program state w.r.t. a particular grace period (Part VII).

In the rest of this chapter, we claim and justify the novelties that this thesis brings on each of the constituents, i.e., *proof* and *specification*, of program verification.

1.1 An Overview of OS Verification

Operating system kernels, as a widely used software system composed of multiple different components, have been a well-known application ground for these verification techniques [30,71–73,93,163]).

Although the prior works on OS verification provide more understanding of many challenging aspects of complete real OS verification [73,93], the proof they present in these verification efforts relies on certain simplifications (assumptions) which we think are crucial for the validity of the proof presented.

1.1.1 The Issue of Memory Virtualization in the Context of OS Verification

One of the biggest simplifications is made on virtual-memory-address translation and management. Virtual memory management lies at the core of modern OS kernel implementation. It is deeply

connected with most other parts of a typical general-purpose OS kernel design, including scheduling, hardware drivers, and the filesystem buffer cache. In writing the authoritative reference on the internals of the Solaris kernel, McDougall and Mauro went so far as to claim that "*the virtual memory sub-system can be considered the core of a Solaris instance, and the implementation of Solaris virtual memory affects just about every other subsystem in the operating system*"[119]. This makes the verification of the virtual memory management subsystem of an OS kernel critical to the correctness of every other piece of the kernel or any software running on top of it.

Virtualization of memory addresses intuitively can be thought of as the mechanism providing more memory resources than the computing machine has. The memory locations (addresses) seen by most code are not, in fact, the exact location in physical memory where the data reside. The mapping between these exposed *virtual* memory addresses and actual physical resources is handled by the cooperation of hardware and OS. Ensuring virtualization memory locations establishes one of the most important aspects of systems software design, *isolation*. Concretely speaking, with memory virtualization, we can realize the separation of process memory resources: the OS manipulates hardware functionality to ensure that any attempt by a process to access memory not explicitly granted to it by the kernel will fail. However, establishing control over virtualized memory address mappings has multiple challenging aspects. First, any control requires interacting with the designated hardware subsystem (i.e., MMU's tables) which exposes itself as an *in-memory kernel data structure* that enables shared (overlapping) access to physical memory regions.

Another complicating matter is that the addresses are *agnostic* to the process they are used in: they reveal no information about which address space they originate from. Keeping track of which *assertions* hold in different address spaces during kernel verification is difficult: Some assertions should hold across all address spaces, while others hold in only one, and others may hold in multiple, but still not all.

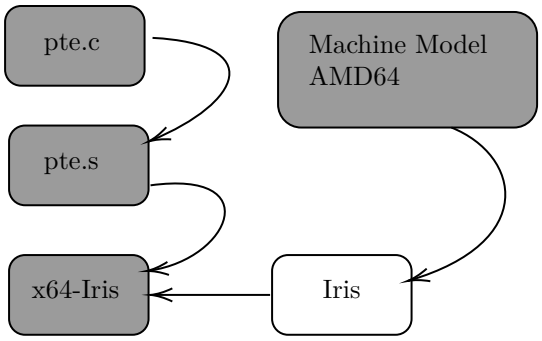


Figure 1.1 Components Showing the Contributions in Gray Boxes

1.1.2 Contributions to Verification in the Presence of Location Virtualization

A context-dependent assertion in which a fact may be true in one address space but not others has a modal flavor. As part of this thesis, we propose modal abstractions to express the truth for the systems (e.g., virtual memory subsystems), which allows us to label assertions true under *other, named* circumstances (i.e., in another address space) with a modality indexed by a name for that space (in our case, the root of the page tables for an address space). This offers a *convenient* and *powerful* way to *modularly* isolate assertions specific to a particular address space, explicitly state when an assertion is true across address spaces, manipulate address spaces from within other address spaces, and reason about changes in address spaces. By exploiting the expressiveness of our abstractions for virtual-memory management, we can treat the virtual memory reasoning more flexibly than prior program logic techniques[97,98], which could only work with a single address space (the current address space on the CPU) because they were unable to speak directly *within the logic* about other address spaces, and capture the effects of page-table updates within and across address spaces.

New modal abstractions Some of our experiments demonstrate the suitability and flexibility of a modal treatment of address spaces, which we briefly discuss above.

New Foundations for understanding memory addressing We develop these ideas in the form of a logic named x64-Iris with designated propositions (vProp) for working with virtual-address-space-relative assertions, implemented as an embedded separation logic within the Iris[90] separation logic. The result is a separation logic that lifts several major semantic restrictions present in the few prior logics[97,98] dealing with virtual address translation. The logic we develop covers core reasoning principles for reasoning about memory configurations and code reliant upon or manipulating those memory configurations in the presence of in-memory page tables, the primary memory protection mechanism across Intel/AMD's x86-64 processors, ARM's application class processors including AArch64 CPUs, POWER, RISC-V, and other architectures.

Soundness against more realistic address-translation machine model The soundness of our logic is composed of the proofs of specifications of instructions in our RISC-like fragment of AMD64 whic themselves are in type vProp. The crux point that appears in our soundness proof is the reflection of a realistic address translation in our model onto the proofs of instruction including memory accesses.

Verifying a core kernel memory management code We verify simplified versions of several critical virtual-memory-related pieces of OS functionality, including mapping pages, switching address spaces, and converting physical addresses to virtual addresses for software page table walks (as when mapping pages). For the experimental setting, we take our kernel's core virtue-memory-management unit (pte.c) which utilizes address translation (with 4 levels of page-table tree) for locating physical pages mapping virtual addresses. We verify the assembly instructions produced by C compilers.

Our examples presented for *memory-virtualization* constitute the core of memory management code. We present a concise specification, and each example either goes beyond the technical capabilities of prior logic or revisits an example from prior work with more details (treating parts of the kernel invariant never addressed by prior work) and fewer assumptions (e.g., verifying virtual-to-physical conversion rather than assuming it has already occurred).

1.2 An Overview of Protocol Based Reasoning

An important and challenging aspect of verifying computer systems is the migration of a proof already made against a specification to a new one. Specifications, at their core, form a protocol on the access to the state of a data structure, a subsystem of computing systems, or multiple client machines *interfering* on updating a server's shared state. When we think of a simple protocol for handling file usage, we would expect the file to be opened before we read or write to it.

State transition systems have been used to encode these protocols as part of program logics, specifically within concurrent program logics, to specify the coordination of threads to cooperate in the modification of shared state[53,91,139,156,157]. They encode an STS by giving a set of abstract states. Each abstract state is associated with an invariant and the invariant applies to the actual program state that corresponds to a particular abstract state.

To verify that a mutation moves the data structure along the protocol correctly, each update from a given abstract state must modify the program state to match a permitted destination state's invariant, including token ownership[1]. For example, modeling a basic mutual exclusion lock in this way includes two abstract states **locked** and **unlocked**, and two tokens lock and unlock. In the **locked** state, the data structure's representation includes ownership of the token unlock; unlocking the structure (releasing the lock) requires a state update to match the invariant of **unlocked**, which requires transferring the unlock token into the data structure, were a subsequent lock acquisition will obtain it.

This class of specifications is flexible, intuitively appealing, and with a modest extension, permits adapting the long-standing idea of specifying a system with a state machine to support concurrent clients which induces a natural variant of rely-guarantee reasoning[87]: a client is limited to those transitions enabled with the capabilities it owns, and interference from concurrent clients is limited to at most those transitions made possible by the *complement* of the client's capabilities.

[1] We are simplifying slightly; the knowledge of the origin and destination may be imprecise, and therefore this check is done for *each possible* initial and final abstract state.

Motivation for Foundational Approach to Understand the Modularity of State-Transition-Systems STSes offer a concise, high-level specification for the interactions between concurrent clients of a data structure, subsystem, or server, and those communications' effects on data structures. However, modularity for STSes has not been thoroughly studied. Early systems lack modular STSes[156] beyond simple nesting (e.g., the invariant for one STS state referring to ownership of another STS's state), achieve limited forms of modularity as a consequence of working with impredicative higher-order separation logics (which offer limited forms of qualification and therefore subsumption)[149], or are limited to essentially a product construction over STSes where code verified against one component can be used with a product containing it[144] (this last is a simplification we will revisit in detail later).

This is a problem, because real systems are full of implicit protocols, but with differences between them that current approaches to modular STSes are ill-suited to support. Consider, for example, the various layers of filesystem abstraction in an operating system kernel. The kernel specifies a range of operations on files, but most application code uses only a small subset of them (opening, reading, writing, and closing). This is especially important for cross-platform code: different kernels offer different operations (and some offer different semantics for common operations!) so applications code to the common (consistent) subset. Supporting such verification requires one to be able to abstract kernel-exposed file protocols in such a way that clients can ignore certain operations and file states. At the same time, the kernel requires the lower-level filesystem drivers to implement a range of operations sufficient for the kernel to implement the full range of operations it dictates (even if client programs ignore many of them). However, some operations make no sense for some filesystem. Consider an in-memory filesystem backed by a chunk of RAM: There is no sensible notion of syncing such a filesystem with the disk because by design it is not backed by the disk. So to verify that an in-memory filesystem adheres to the protocol required by the kernel — which would specify separate abstract states for synced and unsynced data — we would need to abstract the protocol obeyed by the in-memory filesystem in such a way that it adds additional states and transitions that map onto the ones that exist from the filesystem's perspective. Aside from ad-hoc means, current approaches

do not support the required abstractions where we relate two protocols that differ by additional states and transitions, and a verification made against one can be adapted to the related one. Those that support one kind use mechanisms where the sorts of permissible abstraction must be planned for upfront.

1.2.1 Contributions to Modularity of Protocols

New foundations on STS reasoning In this thesis we introduce a single form of abstraction for STS specifications that support both hiding states and operations (as with an application ignoring operations the kernel permits) and fabricating states and operations. We extend and adapt the classic notion of bisimulation (in the modal logic sense) by treating STSes essentially as a specialized form of Kripke structure [20,102].

Linking to a program logic Then, we introduce a proof rule that permits this form of "subtyping" on STSes to be formalized as an extension to the Iris[91] formalization of protocols. Our soundness argument for STSes, an extended version of the one in CaReSL, relies on the preservation of the bisimulation relation we introduce for STSes against admissible logical updates on STSes which themselves are logical assertions.

Experimenting on file protocols Finally, we show how our bisimulation rule relates to a conventional file protocol that enforces the protocol of writing to only an opened file where updates can be observed synchronously to a distributed file protocol that differs in propagating the write over a network and requires explicit synchronization to observe the writes locally.

1.3 An Overview of Concurrent Memory Management

Reasoning about concurrent programs is hard. We need to consider all interactions between threads because side-effects in one thread can affect the behaviour of another thread. When we consider non-blocking algorithms, reasoning becomes harder because interactions between threads running non-blocking algorithms are more subtle than the ones exist in lock-based mutual exclusion. There are logics and automated reasoning tools to address challenge in reasoning consistency of non-blocking algorithms.

Contingency: Relatively Consistent Memory View Normally, doing a set of memory operations simultaneously or indivisibly may end up having side-affects and a partially completed set of operations, and may result in inconsistency for the clients of the data structure.

However, there are many cases where strict consistency may be relaxed with temporal inconsistency if clients may tolerate stale data. This is mainly done with delaying any work that is not urgent to be performed. This relaxation brings new kind of programming pattern. Mutator processes' accesses to shared data are mutually exclusive via writer lock that prevents simultaneous update of a node by more than one process. In this concurrent setting, there is a set of reader processes and a set of updater processes with a process or set of processes to reclaim memory locations where nodes in the free list reside. An example of delaying of work occurs in garbage collection of heap locations. It is generally unnecessary to collect garbage heap locations immediately after it is generated, [103].

Implementing Relatively Consistent Memory Management 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 [126] 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) [120] 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[10,32,48,103,121,155], and RCU plays a key role in Linux kernel memory management[124].

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[3,47,95,114], but this does not guarantee correctness of arbitrary client code. Sophisticated verification logics can prove correctness of the RCU primitives and clients[63,70,89,118,153]. 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[125] 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.

1.3.1 Contributions to Verification of Clients using Relatively Consistent Memory Management

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.

1.4 An Overview of Modal Verification Patterns

Low-level systems exhibit certain patterns in their designs, especially when interacting with computing resources. Exploiting certain patterns while designing software has been an important field of study. In this regard, we think that certain properties of modalities enable us to understand and do the verification challenges that show certain patterns.

1.4.1 Contributions to Identifying the Verification Patterns via Modal Abstractions

We argue how modal abstractions can be used to identify and abstract system verification challenges. We justify our perspective by discussing prior systems that have successfully used modalities for system verification, arguing that they fit into the verification design pattern we articulate, and explaining how this approach might apply to other systems' verification challenges.

Identifying System Verification Challenges We start with identifying common patterns in system verification: *virtualization, sharing, and translation.*

Introducing the Concept of *Resource* Then we discuss the concept of *resource* which has already been an essential concept in the design of systems. Inspired by the concept of resource in the systems, we define what a resource and its context are in the modal abstractions.

Introducing the Concept of *Nominals* Nominalization enables identifying a resource in a context. For example, a transaction is a context of resources of in-memory updated disk blocks. The transaction identifier is used to associate a transaction with a disk-block to be persisted so that, in case of a crash while persisting updated disk-blocks, the filesystem can rollback the *already persisted* disk-blocks of the transaction, and reach to the previous consistent disk state. To be able to do so, both the updated in-memory disk blocks and the transaction must refer to the transaction identifier – *strong nominalization.* In another example, virtual memory references (resources) in an address space (resource context), which can be uniquely identified with a root address (the nominal)

of its page-table tree, are *agnostic* to the address space that they are in. However, they can only be accessed (be valid) in the address space to which they are agnostic. However, then an address space switch happens, the virtual memory references of the previous address space must be made inaccessible. To be able to do so, although virtual memory references do not hold any piece of information related to their address space, we still have to associate them. We call this kind of unilateral nominalization – *weak nominalization*.

Taxonomy of the Current Modal Approaches in System Verification Based on these concepts defined, we summarize contemporary verification efforts using modal abstractions. We choose them from different domains, for example, weak memory verification, storage persistence [27,28,40,56,57,106,154,160,162] because we would like to justify that our *definitions* are not domain dependent.

1.5 Reading Guideline

In this section, we give the outline of the thesis to facilitate reading. Except for the first part, which gives an overview of each of the following parts separately, each part starts with a background knowledge section, and the accumulation of the background knowledge introduced in each part should be assumed to be enough for reading the rest of the thesis.

Part I gives an overview of each of the parts following separately in different subsections.

Part II starts with giving background knowledge on the separation logic Iris and briefly mentions the modal aspect of *location virtualization*. Then, it introduces the reasoning principles and discusses the proofs and experiments.

Part III starts with giving background knowledge on protocol-based reasoning and discusses the modal aspects of it. Then, it introduces the bisimulation relation in two layers: the core principles and then the behavior invariants in the bisimulation. It discusses how the bisimulation relation is linked and utilized in Iris and finally shows the migration of the proof of a file-write operation

between different protocols.

Part IV starts with explaining the concurrent memory management model, which itself exhibits contingency on the consistency of the program state. Then, it introduces the reasoning principles and finally discusses the application of them in the experimental setting.

Part V presents the definitions that we use to identify the verification patterns according to the modalities used. Then, as an experimental setting, it takes the recent system verification efforts using modal abstractions and identifies the pattern they follow based on the given definitions.

Acknowledgements Please note that a part of the machine model was implemented as an undergraduate research project by Austin Herring. The text of Parts II and V is based on draft papers co-authored with my advisor. The text of Part IV is based on the paper published in ESOP'19 [105] and its technical report [104].

Part II

Modal Abstractions for Location Virtualization

CHAPTER 1

BACKGROUND

In this chapter, we give background knowledge on the logical principles that we use to build our x64-Iris atop. Our reasoning principles are mainly constructed on top of program logic called *separation-logic* (SL). In this thesis, we use a realization of SL called Iris. Therefore, we give a brief background knowledge on the logical constructions we use from Iris.

1.1 Program Logic

A methodology that enables specifying and proving programs is called *program logic*. The validity – *soundness* – of what a program logic offers is proven against a model: a machine model with a state (e.g. heap, stack). Hoare Logic is one of the well-known, simple program logic for sequential programs. Hoare Logic's specification form has almost been a standard followed by other program logic

$$\{P\}\ e\ \{Q\}$$

interpreted as *"if a program expression e is run in a state satisfying P, and it terminates, then the state obtained after e is run satisfies Q"*. We call the predicate P and Q the precondition and the postcondition respectively. To reason about larger programs, Hoare Logic offers structural *proof-rules* such as sequencing (Seq) to compose specifications of different parts (e1 and e2) of the larger (e1;;e2)

$$\textsc{Seq}\quad \frac{\{P\}\ e1\ \{Q\} \qquad \{Q\}\ e2\ \{T\}}{\{P\}\ e1;;e2\ \{T\}}$$

1.2 Background on Iris Separation Logic

Separation logic (**SL**) [132] is a special treatment of Hoare logic with some specialized proof rules and assertions for resources.

1.2.1 Basic SL Assertions

This special treatment starts with introducing specific assertions to speaking to resources such as heap. The essential assertion to talk about the resources is *points-to* relation: $p \mapsto v$ asserts the fact that the pointer p when dereferenced has value v.

The other crucial special treatment is *separating conjuction* $*$. $P * Q$ asserts that P holds on a piece of heap that is disjoint from where Q holds.

In an intuitionistic separation logic, a heap assertion that is true in one heap is also considered to be true in the larger one. The entailment between propositions is written $P \vdash Q$, read as "P entails Q", which says that in any heap where P holds, Q must also hold.

$$\textsc{Mono}\quad \frac{P \vdash P' \;\; Q' \vdash Q \qquad \{P'\}\ e\ \{Q'\}}{\{P\}\ e\ \{Q\}}$$

Separating conjunction, $*$, together with the proof rule Mono, immediately exhibits the flavor of locality we observe in the proof rule called Frame. This rule informally states that computation for a triple $\{P\}e\{Q\}$, executing e in a state satisfying P does not change the state in a way that it is no longer satisfying P.

$$\textsc{Frame}\quad \frac{\{P\}\ e\ \{Q\}}{\{P * R\}\ e\ \{Q * R\}}$$

The Simplest form of Concurrency in SL The justification we make for the validity separating conjuction (Frame rule) not only brings the *locality* intuition but also opens a room for a basic

$$\frac{a \rightsquigarrow_{M_i} B}{\boxed{a : M_i}^{\gamma} \Rrightarrow \exists b \in B.\, \boxed{b : M_i}^{\gamma}}\ \textsc{GhostUpd} \qquad \frac{}{\boxed{a : M_i}^{\gamma} * \boxed{b : M_i}^{\gamma} \leftrightarrow \boxed{a.b : M_i}^{\gamma}}\ \textsc{GhostEq}$$

Figure 1.1 Overview of Proof Rules for Ghost Resources [91]

concurrency in which no sharing of resources allowed (Rule Parallel)

$$\textsc{Parallel}\ \frac{\{P1\}\ e1\ \{Q1\}\quad \{P2\}\ e2\ \{Q2\}}{\{P1 * P2\}\ e1||e2\ \{Q1 * Q2\}}$$

However, most of the useful concurrent programs includes some sort of *sharing*. Unfortunately, Parallel rule, which relies on the disjointness of resources ($P1 * P2$), cannot not be used in realistic concurrent programs.

1.2.2 Abstracting SL

SL generalization frameworks [50,91] enabled logical mechanisms to construct custom-tailored separation logics for different verification challenges. Iris, which we use in thesis, is also an abstract **SL** framework [91]. Iris provides two key mechanisms to the problems we mentioned above: the concept of custom-tailored resources on top of partial-commutative-monoids (PCM)s, and invariants for knowledge-sharing.

Customized Assertions with Named Ghost Resources

(**PCM**) provides the convenient algebraic structure to construct *composable* resources as elements of **PCM**s. These resources are logical (abstract) resources to augment the physical resources (e.g. memory, stack, registers etc.) – no effect on the program's physical state (execution) but help with abstracting the effects of local state, but also additional bits for easing verification (e.g. concurrency) in a uniformed fashion.

A ghost resource $\boxed{a : M_i}^{\gamma}$ asserts the ownership of a part a of the instance named γ. In Figure 1.1, we see a set of rules for utilizing the notion of ghost resources in Iris [91]. Owning $"a"$ of ghost

resource is called *fragmental* ownership. Knowing the non-conflicting composition of the picked up operator, . in Rule GhostEq, the complete global ghost state is composed the fragmental ownerships related to the resource. In addition to the ability of speaking on its (resource's) structurally (by splitting/composing), we can also concretely utilize them via asserting the change on them. Since ghost resources are augmentable to the physical state (i.e. change on them does not affect the physical state), one can pick an arbitrary global physical state and update the ghost resource with *view-shift* operator, $P \Rrightarrow Q$: admissible to update the resources (derived Rule GhostUpd in Figure 1.1) satisfying P to satisfy Q without changing the underyling physical state. The obvious utilization of view-shift appears as an encoding of entailment ($\vdash$ in Rule Mono) in the consequence rule Csq

$$\textsc{Csq}\quad \frac{P \Rrightarrow P' \;\; Q' \Rrightarrow Q \qquad \{P'\}\ e\ \{Q'\}}{\{P\}\ e\ \{Q\}}$$

On top of ghost resources, Iris offers *authorative* ghost resources. Authoritative ghost resources allow you to distribute the fragmental ownerships of a resource, and bookkeep these fragments within the authoritative state to make sure the soundness (validity) of accesses to the resource is preserved. We pervasively use this style of reasoning as part of our principles for abstracting page-table-walks in Section 4.3. Fragmental ownership of the abstract page-table-walk is given to the virtual memory translation resources (called them virtual-pointsto relations), and we ensure the soundness (validity) of a virtual address translation because *other* virtual address resources also have fragmental ownerships which, on its own, is not enough to change the state of the resource abstracted – i.e. physical page-table.

Protocol Style Reasoning

Although we already get the flavor of how to utilize fragmental ownership of ghost resources, it is still not clear how they interact with the full ownership of the same ghost resource. One might have already noticed that fragmental ownership of a ghost resource allows you to assert a *local view* on a

resource. On the other hand, the authoritative ownership regulates the access to the *global-view* of the resource. The core piece of assurance inherent from using the named monoids lies in its pointwise lifting of the composition – i.e. the combined fragments exceeding the combined authoritative element or having two authoritative elements composed cannot co-exists. Co-existence of an authoritative element with a fragmental one is defined over the *cancellativity* of the monoids: the property that asserts the ownership of the authorative elements' capability for exchanging the owned fragmental with one another compatible one. These two principles (pointwise lifting of the composition and cancellativity) constitute the essence of a ghost resource as an instance of authoritative monoid [91].

Invariants With Iris's logical constructions we mentioned so far, we can only express how accesses to resources – owned either one thread or another – can be done. *Sharing* a resource between multiple parties, is realized with the logical construction called *invariants* [91]. Invariants have the assertion form $\boxed{\mathrm{P}}^n$ which expresses the shared knowledge (named n) that there are resources satisfying P. This fact (shared knowledge) is valid because it can only be constructed once each thread has an agreement on it, i.e. all threads agree on the state of the resources satisfying P. Therefore, when a thread would like to utilize the shared knowledge while discharging the obligation in the spec of an atomic action (α), it first opens the invariant $\boxed{\mathrm{P}}^n$ in its precondition, then use P just for the α as a single program step, and, finally, for the sake of not invalidating other threads knowledge on P, it has to be reestablished.

$$\textsc{Inv}\quad \frac{\{P * R\}\ \alpha\ \{P * Q\}_{\epsilon} \qquad \alpha \text{ physically atomic}}{\boxed{P}^n \vdash \{R\}\ \alpha\ \{Q\}_{\epsilon \uplus \{n\}}}$$

1.3 Modalities in Logic

Assertions of modal logic express the contingent truth. This contingency could be temporal (temporal logics [137]) which can express facts holding *always or eventually*; or could be about knowledge and belief [77,81] etc. Our inspiration for our modal address-space is partially rooted in dynamic logic

[79,138] satisfaction operator: $[\![r]\!]P$ states that P holds on the resource context (i.e. address space abstraction as a bag of virtual-to-physical address mappings) rooted at r. Although there has been recently increasing interest in using modal operators directly as part of reasoning principles, mostly for weak-memory pending updates [146,157], they had always been essential in incorporating the capabilities for handling issues such as impredicativity as a logical mechanism inside a program logics [91,100,149] or type systems [9,16,17,127].

CHAPTER 2

RELATED WORK

2.1 Verification Effort on VMM

There has been relatively little prior work on formal verification of virtual memory. Instead, most OS verification work has focused on minimizing reasoning about virtual memory management. The original VERISOFT project[5–7,7,39,80,147] relied on custom hardware which, among other things, always ran kernel code with virtual memory disabled, removing the circularity that is a key challenge of verifying VMM code for real hardware: at that point page tables become a basic partial map data structure to represent user program address translations, with an idiosyncratic format. It turns out that subsequent OS verification work also treats page tables this way, but unsoundly given that other projects target hardware that *does* run the kernel with address translation: they *trust* that the particular page table manipulations do not, for example, unmap kernel code (which can crash the machine even if done "temporarily"[1]).

SEL4[93,94,145] is a formally verified L4 microkernel[115,116] (and the first verified OS kernel to run on real-world hardware), verified with a mix of refinement proofs and program logic reasoning down to the assembly level. Because SEL4 is a microkernel, most VMM functionality actually lives in usermode and is unverified, and moreover, their hardware model omits address translation entirely and the MMU entirely[93,94]. As a result, the limited page table management present in the microkernel treats page tables as idiosyncratic tree-maps as in VERISOFT, despite actually running with address translation. This is partly mitigated by manually identifying some trusted invariants (e.g., that the

[1] A bug one author has personally encountered in a research kernel, though not one discussed here.

address range designated for the kernel is appropriately mapped) and setting up the proof to ensure those invariants are maintained (i.e., as an extra proof obligation not required by their hardware model).

CertiKOS[30,72–74] is a microkernel intended for use as a hypervisor. The overall approach in that body of work is many layers of refinement proofs, using a proliferation of layers with small differences to keep most individual refinements tractable. In keeping with precursor work on the project from the same group[159], the purpose of some layers is to abstract away from virtual memory (as early as possible). The papers on CertiKOS do not explicitly detail the VMM beyond highlighting its existence and referencing that it performs mapping operations for user code. The work is clear, however, that it fully trusts low-level assembly fragments such as the instruction sequence which actually switches address spaces, rather than verifying them. Another key aspect of their approach is that the OS is written in Clight and compiled with CompCert[21,110,111]. CompCert's memory abstraction[111] assumes memory is a set of disjoint chunks of bytes with no overlap, so the lowest levels of CertiKOS must provide a matching machine model as a layer. This prohibits virtual address aliasing, so CertiKOS cannot support simultaneous memory-mapped (`mmap`) and stream-oriented (`read`/`write`) IO to a single file, and cannot use the common kernel design choice of mapping all physical memory into the bottom of the kernel's address space for direct access i while the kernel code is simultaneously mapped (and executed) at higher virtual addresses. This is not necessary for CertiKOS's intended primary use case (a hypervisor), but means that CertiKOS's approach cannot be used to support this functionality in other systems, without major surgery to CompCert.

Other work on OS verification either never progressed far enough to address VMM verification (Verisoft XT[33,34,36,38]), or uses memory-safe languages to enable safe co-habitation of a single address space by all processes (Singularity[14,59,85,86], Verve[163], and Tock[112]).

Some work has been done like ours, studying VMM verification separately from the rest of the kernel. One important outgrowth of the seL4 project, not integrated into the main project's proof, was work by Kolanski and Klein which studied verification of code against a hardware model that *did* include

address translation — the only work aside from ours to do so — initially in terms of basic memory[97] and subsequently integrating source-level types into the interpretation[98]. They were the first work to model physical and virtual points-to assertions separately, defining virtual points-to assertions in terms of physical points-to assertions mimicking page table walks, and defining all of their assertions as predicates on a pair of (physical) machine memory and a page table root, an approach we improve on.

They also define their virtual points-to assertions such that a virtual points-to $p \mapsto_{\mathsf{v}} a$ owns the full lookup path to virtual address p. This means that given two virtual points-to assertions at the same time, such as $p \mapsto_{\mathsf{v}} a * p' \mapsto_{\mathsf{v}} b$, the memory locations traversed to translate p and p' must be disjoint. This means the logic has a peculiar limit on how many virtual points-to assertions can coexist in a proof. Since page tables fan out, the bottleneck is the number of entries in the root table. For their 32-bit ARMv6 example, the top-level address is still 4Kb (4096 bytes), and each entry (consumed entirely by a virtual points-to in their scheme) is 4 bytes, so they have a maximum of 1024 virtual points-tos in their ARMv6 configuration. Any assertion which implies more than that number of virtual addresses are mapped implies false in their logic. (They do formulate their logic over an abstract model, but every architecture would incur a similar limitation.) Our definitions make use of fractional permissions throughout; Figure 4.1's definition of L4_L1_PointsTo elides the specific fractions used, but it in fact asserts 1/512 ownership of the L1 entry, $1/(512^2)$ of the L2 entry, and so on, so each entry may map the appropriate number of machine words.

As noted earlier, by collocating both the physical ownership of the page table walk as part of the virtual points-to itself these logics preempt support for changes to page tables which do not actually affect address translation. We address this by moving from Figure 4.1's definition (essentially Kolanski and Klein's with fractional sharing of intermediate table entries) to Figure 4.1's.

Kolanski and Klein do verify code to map in a new page by installing an L1 page table entry, akin to our Figure 5.5. However, our logic treatment of mapping goes beyond theirs. Kolanski and Klein must unfold machine semantics at one point in their proof of mapping correctness[97] p. 27, while our

logic permits us to conduct the proof entirely in separation logic. Their proofs also do not address converting physical addresses of page table entries to virtual addresses that can be used to access them. Their original low-level (pseudo-assembly) proof assumes the correct virtual address already exists[97], and their subsequent C-level proof[98] axiomatizes a function akin to our ensure_L1. While full proof of ensure_L1's page table walk is ongoing work, we have verified that virtual-to-physical translation can be done in our logic with appropriate kernel invariants (Section 5.1), while Kolanski and Klein's model may not be able to do so — because the virtual points-to owns the physical resources associated with intermediate page table entries, it appears impossible to have a virtual points-to for a page table entry that is part of the page table walk for another virtual address.

The other major distinction is that Kolanski and Klein have no explicit accounting for other address spaces. Their logic does not deal with change of address space, and has no way to assert that certain facts hold in another address space. They verify only one address space manipulation: mapping a single unmapped page into the current address space (in both papers). We verify this, as well as a change-of-address-space, which requires us to introduce assertions for talking about other address spaces (we must know, for example, that the precondition of the code after the change must be true in the *other* address space), and to deal with the fact that the standard frame rule for separation logic is unsound in the presence of address space changes and address-space-contingent assertions. Our approach in this thesis uses modalities to distinguish virtual-address-based assertions that hold only in specific address spaces, making it possible to manipulate other address spaces, and equally critically, to *change* address spaces while reasoning about correctness.

Unlike our work, Kolanski and Klein prove very useful embedding theorems stating that code that does not modify page table entries can be verified in a VM-ignorant program logic, and that proofs in that logic can be embedded into the VM-aware logic (essentially by interpreting "normal" points-to relations as virtual points-to facts). While we have not proven such a result, an analagous result should hold of our work: consider that the doubles for the `mov` instructions that access memory behave just as one would expect for a VM-ignorant logic[31]. With our general approach to virtual points-to assertions being inspired by Kolanski and Klein, *both* our approach and theirs could in

principle be extended to account for pageable points-to assertions by adding additional disjunctions to an extended points-to definition; embedding "regular" separation logic into such a variant is the appropriate next step to extend reasoning to usermode programs running with a kernel that may demand-page the program's memory.

2.2 Modal Abstractions in Systems Verification

As noted earlier, the inspiration for our other-space modality comes from hybrid logic[11,18,65,66], where modalities are indexed by *nominals* which are names for specific individual states in a Kripke model. We are aware of only two prior works combining hybrid logics with program logics specifically. Brotherston and Villard[24] demonstrated that may properties true of various separation logics are not definable in boolean BI (BBI), and showed that a hybrid extension HyBBI allows most such properties to be defined (e.g., the fact that separating conjunction is cancellative is unprovable in boolean BI, but provable in HyBBI). There, nominals named resources (roughly, but not exactly, heap fragments). Gordon[67] described a use of hybrid logic in the verification of actor programs, where nominals named the local state of individual actors (with such assertions stabilized with a rely/guarantee approach). Beyond these, there is limited work on the interaction of hybrid logic with substructural logics, in restricted forms that do not affect expressivity. Primarily there is a line of work on hybrid linear logic (HyLL)[49], originally used as a way to more conveniently express aspects of transition systems in linear logic. However, HyLL's proof rules offer no non-trivial interactions with multiplicative connectives (every HyLL proof can in fact be embedded into regular linear logic[29], unlike Brotherston and Villard's HyBBI, which demonstrably increases expressive power over its base BBI.

In both HyLL and HyBBI, nominals denote worlds with monoidal structure (as worlds in Kripke semantics for either LL or BBI necessarily have monoidal structure). Our nominals, by contrast, do not name worlds in the same sense with respect to Iris's CMRAs, but in fact *classes* of worlds, because the names are locations (a means of *selecting* resources) rather than resources. A key difference is that the use of nominals in those logics corresponds specifically to hypothetical reasoning

about resources (until a nominal is connected to a current resource, in which case conclusions can be drawn about the current resource), which means the modalities themselves do not "own" resources. Instead, assertions under our other-space modality can and do have resource footprints. Pleasantly, we sidestep most of the metatheoretical complexity of those other substructural hybrid systems by building our logic within a substructural metatheory (Iris).

Iris has been used to build other logics through pointwise lifting, notably logics that deal with weak memory models[41,42]. Those systems build a derived logic whose lifting consists of functions from thread-local views of events (an operationalization of the release-acquire + nonatomic portion of the repaired C11 memory model[107]): there modalities $\Delta_\pi(P)$ and $\nabla_\pi(P)$ represent that P held before or will hold after certain memory fence operations by thread π. The definitions of those specific modalities existentially quantify over other views, related to the "current" view (the one where the current thread's assertions are evaluated), and evaluate P with respect to those other views. This approach to parameterizing assertion semantics by a point of evaluation, and evaluating modalized assertions at other points quanfied in the definition of a modality, is the classic notion of modal assertions, whereas hybrid logics expose the choice of evaluation point in assertions, allowing statements of more properties. In these weak memory examples this additional expressive power would not be useful, because any relevant points of evaluation (thread views) are intimately tied to memory fences performed by the program, whereas for virtual memory management the kernel must be able to choose or construct arbitrary other address spaces.

CHAPTER 3

SEMANTICS

3.1 Overview on Machine Model

In typical system configurations, all memory addresses seen by programs running on modern computers are *virtualized*: the address observed by a running program generally will not correspond directly to the physical location in memory and may not even correspond to a physical location that *exists* in the machine. Instead, these *virtual* addresses are translated to *physical* addresses that correspond directly to locations in RAM. On most modern architectures, this translation is performed through cooperation of the hardware and OS kernel: while executing an instruction that dereferences a (virtual) address, the CPU's *memory management unit* (MMU) hardware performs *address translation*, resulting in a physical address used to access the cache[1] and/or memory-bus.

On the x86-64 architecture, the MMU's address translation uses a sparse hierarchical set of tables: *page tables* (referring to pages of memory). As Figure 3.1 (based on Figure 5-17 of the AMD64

[1]Technically, for performance reasons most caches are indexed with parts of the virtual address, but tagged with the physical data addresses, so cache lookups and address translations can proceed in parallel.

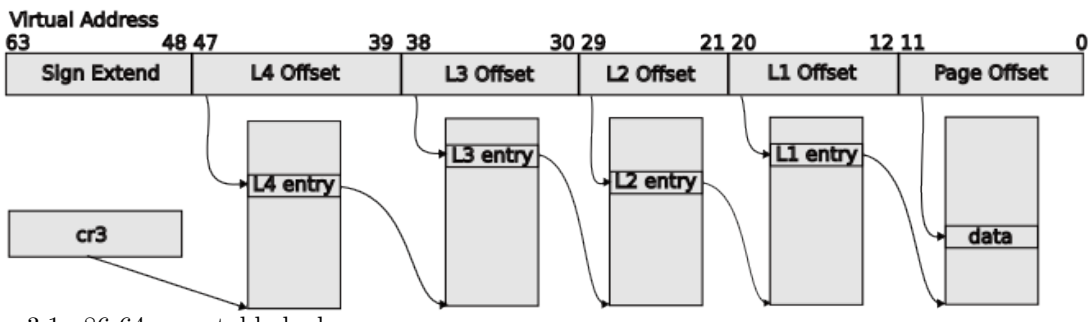


Figure 3.1 x86-64 page table lookups.

architecture manual[82]) shows, address translation proceeds by repeatedly taking designated slices of the virtual address and indexing into a table. The final lookup in the page tables gives the base physical address of a 4KB page of physical memory, to which the low-order bits of the accessed virtual address are added to determine the actual physical address retrieved. On x86-64, standard configurations use 4 levels of page tables, labelled levels 4 through 1, with lookups in the level 1 page table resulting in the actual page of physical memory holding the requested data, and the low-order 12 bits being used to index into this page.[3] The translation process or algorithm is sometimes referred to as a *page-table walk*. While Figure 3.1 and most of our constants (how many levels, which virtual address bits index which table levels) are specific to the x86-64 architecture, ARMv8 (a.k.a. `aarch64`), RISC-V, and PowerPC use similar hierarchical page tables for address translation. RISC-V's `sv48` paging configuration[1] §4.5 and AArch64's 4-level paging configuration both split 64-bit virtual addresses at the same points to index into the respective tables, so most of what follows is equally applicable to those architectures (the only difference is that page table entries place control bits of write access, etc., in different orders).

The entries of each table are 64 bits wide, but each points to a physical address aligned to 4KB (4096 byte) boundaries, which leaves 12 bits to spare to control a validity bit (called the *present* bit), a read-write bit (which permits write access through the entry if and only if it is set), and a range of additional bits that can be used to control caching, write-through, and more. This thesis will only consider the present bit (0).

The page tables are managed by the OS, typically by a *virtual memory manager* (VMM).[4] Typically each process has its own page table, which the OS registers with the CPU by storing the (page-aligned) physical address of the root of the page table tree showing the start of the L4 table) in a specific register (`cr3`) as part of switching to a new process. Using different mappings, which map only

[2]While x86 up through its 32-bit incarnation were due to Intel, the x86-64 architecture as a 64-bit extension to x86 was originally due to AMD. As a result, it is sometimes also referred to as the `amd64` architecture.

[3]Technically levels 1–3 have explicit historical names, but for brevity and consistency, we simply number them, in keeping with the newer 5th level. Our formalization only deals with 4-level page tables, but is straightforwardly extensible to 5.

[4]Not to be confused with Virtual Machine Monitor. We focus on non-hypervisor scenarios, but hardware virtualization extensions for both x86-64 and ARM make use of an additional set of page tables translating what a *guest* considers to be its (virtualized) physical memory to actual physical memory. Our contributions should offer value in this scenario as well.

disjoint portions of physical memory (with some exceptions in the next section) is how the OS ensures memory isolation between processes.

If an instruction is executed that accesses a virtual address that either has no mapping or does not have a mapping permitting the kind of access that was performed (e.g., the instruction was a memory write, but the relevant address range was marked read-only in the relevant page table entry), the hardware triggers a *page fault*, transferring control to a *page fault handler* registered with the hardware by the OS, allowing it to take corrective action. If no mapping was supposed to exist, this is a program bug (e.g., dereferencing virtual address 0 / NULL) and the faulting program should be terminated. But this can also be used for specialized functionality and optimizations, such as *paging* (saving room in physical RAM and deferring unnecessary IO by only reading program code from disk when it is accessed, or even swapping memory that has not recently been accessed to disk, to read back in when a page fault indicates access).

The key pieces of VMM functionality are adding a new page mapping (whether the mapped page contains zeros, file data, or swap data), and removing an existing page mapping. While this initially sounds like relatively modest functionality whose implementation may be complicated by hardware subtleties, correctness of even these basic operations is actually quite intricate. Notably, updates to the page tables are performed as writes to memory — *which are themselves subject to address translation*, and finding the correct page table to update requires converting between physical and virtual addresses. In the case of changing the mappings for the currently active set of page tables, *the OS kernel is modifying the tables involved in its own access of the tables.*

Virtual memory concerns propagate to the OS scheduler, which deals with multiple address spaces, so must keep track of which virtual addresses are valid (and in what way) in which address spaces. Some virtual addresses are valid in only a single address space (e.g., a code address for a particular user-mode process), while others are valid in all address spaces (e.g., kernel data structure pointers). The VMM must maintain some of these assumptions on behalf of the rest of the kernel, for example by guaranteeing that a certain range of virtual addresses (corresponding to the kernel's code and

data) are valid in every address space.

Translation Lookaside Buffers This section explains the challenges of formal reasoning about TLBs in more detail, including why prior verified OS work trusts the flushing operations (and we believe this is reasonable) and why full support for TLB reasoning is a significant, challenging problem on its own, requiring further development before being integrated with general virtual memory reasoning.

As noted earlier, TLB flushes are required only when addresses are removed from a virtual address space, or when changing virtual address spaces. Because this occurs in few places in the kernel (in some, only 3 locations), fully verified kernels including seL4[93,94] and CertiKOS[72,73] trust TLB management. Neither of the aforementioned systems has a hardware model including a TLB, so neither is able to verify TLB management in any form—they *must* trust its operation.

The only place the code itself becomes particularly challenging is in multiprocessor kernels, where not only must the running CPU flush its TLB, but it must send an inter-processor interrupt (IPI) to all other cores to ensure they also flush the relevant ranges of their TLBs. Currently, no formal hardware model exists with sufficient detail to reason about IPIs in full detail. Multicore extensions of seL4[161] and CertiKOS[73] also trust this functionality.

Fully grounded trust for this would require a formal model of how hardware populates ACPI tables and formal verification of ACPI parsing code on general-purpose desktop and server machines, or a formalized and verified parsing of flattened device trees (plus trust that a system was booted with a correct FDT) for semi-embedded systems like single-board ARM and RISC-V machines. Either route would also require detailed treatment of memory-mapped IO-triggering interrupts on other CPUs. This is far beyond any formal model of computer hardware that exists today and far out of the scope of this thesis.

Syeda and Klein[151,152] are the only existing works to address formal verification of TLB management. They extend the work of[96,98,98], and therefore inherit the limitations of that work discussed elsewhere in this paper. Aside from that, their logic primarily tracks a set of mapped addresses and a set of

cached addresses and generally preserves a global invariant (in a global Hoare logic, not a separation logic) that the cached addresses are a subset of the mapped addresses. When this invariant is violated, the logic provides only a Hoare-style backward assignment type rule with unconstrained assertion P, rather than providing structure in the logic for reasoning about how to restore the invariant. Most technical results in the paper focus on proving transparency lemmas, that code that *does not* modify page tables (kernel code outside the VMM, user code) is unaffected by its existence. The one piece of kernel code verified is a sequence of 4 pseudo-instructions for changing address spaces, which include *only* the installation of a new page table root and flushing of the old address space from the TLB.

Adapting Syeda and Klein's global reasoning principles to a separation logic is non-trivial, even though their approach does capture some intuition about TLB reasoning. And as noted above, in the absence of support for significant additional hardware functionality in the hardware model, this would be substantial work for only very limited gains in confidence compared to trusting single-core TLB management. Thus we leave TLB management to our future work plans, where we believe it is best addressed in tandem with substantially richer hardware models than any currently existing ones.

3.2 Syntax

Programs in our logic are instruction sequences $\vec{i}$, which are formed by prefixing an existing instruction sequence with an additional instruction $(i; \vec{i})$: we instantiate Iris with a simple language for streams of instructions, which are modeled in our machine model and explained in detail in the following sections of this chapter 3.3.

The syntax of instructions is given in Figure 3.2. The structural reduction rules for syntactic elements are shown in Figure 3.3 in which we see that single instruction prefix is where the evaluation contex's cursor is filled (StepContext).

$$
\begin{array}{rcll}
\mathsf{w}_n & \in & \mathcal{W}_n & \\
\mathsf{r} & \in & \mathsf{greg} & \\
\mathsf{rv} & \in & \mathsf{regval} & \\
v & ::= & () & \\
\vec{i} & ::= & \mathsf{skip} & \textit{no-op} \\
 & & i; \vec{i} & \textit{sequencing} \\
K & ::= & [] \mid K; \vec{i} &
\end{array}
$$

Figure 3.2 Syntax

$$
\textsc{StepSeqSkip}\quad \mathsf{skip}; i \;/\; \sigma \longrightarrow i \;/\; \sigma
\qquad\qquad
\textsc{StepContext}\quad \frac{i \;/\; \sigma \longrightarrow i' \;/\; \sigma'}{K[i] \;/\; \sigma \longrightarrow K[i'] \;/\; \sigma'}
$$

Figure 3.3 Structural Reduction Rules for x64-Iris Syntax

3.3 Machine State

To develop our core logical ideas, we first define the physical state (σ) with the following pieces:

- CPU ($\sigma.\mathcal{C}$)
- Physical Memory ($\sigma.\mathcal{M}$) : memory maps $\sigma.\mathcal{M} : \mathcal{W}_{52} \rightharpoonup_{\text{fin}} (\mathcal{W}_{12} \rightharpoonup_{\text{fin}} \mathcal{W}_{64})$.
- Registers ($\sigma.\mathcal{R}$) :register maps $\sigma.\mathcal{R} : \mathsf{greg} \rightharpoonup_{\text{fin}} \mathsf{regval}$

from which we mainly deal with the memory ($\sigma.\mathcal{M}$) and the registers ($\sigma.\mathcal{R}$) within the context of logical constructions presented in this thesis.

3.3.1 Registers

Our model includes all x86-64 integer registers (including stack and instruction pointers), as well as cr3 in type creg (for page table roots) and rflags in type freg (for flags set by comparison operations and inspected by conditional jumps), as shown in Figure 3.4.

```
Inductive greg: Type :=
  | ir: ireg → greg
  | sr: sreg → greg
  | cr: creg → greg
  | fl: freg → greg
  | dr: dreg → greg.

(*
  (E/R)FLAGS register.
  Section 3.4.3 of the
  Intel manual (pg. 77).
*)
Inductive freg: Type :=
  | rflags .
```

(a) Register Constructor and Flags

```
(*
  Integer (general-purpose) registers.
  Section 3.4.1.1 of the Intel manual
  (on pg. 74)
*)
Inductive ireg: Type :=
  | rax | r8
  | rip | r9
  | rbx | r10
  | rcx | r11
  | rdx | r12
  | rdi | r13
  | rsi | r14
  | rbp | r15
  | rsp
```

(b) General Purpose Registers

Figure 3.4 Register Component of the State – $\sigma.\mathcal{R}$

3.3.2 Memory

The essential data units in our model are machine *words*. For clarity and ease of representation, we use machine words, $\mathsf{w}_n \in \mathcal{W}_n$, with the subscripts showing the number of bits in a word, for memory addresses, values, and offsets, rather than distinct location types that wrap machine words: w_{12} is a 12-bit word, which can be obtained, for example, by truncating away 52 bits of a 64-bit word (w_{64}).

Throughout the explanation of operational semantics, we type the physical memory locations with mem64 and virtual ones with vmem64 which is associated with a physical root address as shown in Listing 3.1

```
Definition mem64 : Type := W52Map.t ((word 12 → word 64) + MemFail).
Inductive vmem64 :=
    | virtmem (phys: mem64) (root: address).
Definition address := { w: word 64 | aligned w }.
```

Listing 3.1 Physical and Virtual Memory Definitions

However, as defined, it $\sigma.\mathcal{M}$ is not typed in terms of the virtual or physical memory type, i.e. it

treats all memory types of memory locations as uniformly aligned addresses.

3.3.3 Address-Translation

The core aspect of our operational semantics is *address-translation* of virtual memory addresses. The address translation, pictorially shown in Figure 3.1, is realized with a high-level call to the function translate

```
(*
References: Intel Manual, Volume 3A - System Programming Guide Part 1
(September 2016) - Table 4-12 (pg. 4-19, PDF pg. 123) -
Table 4-13 (pg. 4-19, PDF pg. 123)
*)
Definition translate (m: mem64) (root: address) (w: address): address + mem_fault :=
    match translate_top_level m (proj1_sig root) (proj1_sig w) with
      | inr f ⇒ inr f
      | inl a ⇒ inl (exist _ a I)
    end.
```

Listing 3.2 Address Translation Function

with a root address vcalue root that is hold in the control register (cr3), and the virtual address to be translated (w) including indices used for traversing page tables in the memory m ($\sigma.\mathcal{M}$) – shown in Listing 3.3.

Remark 1 (Non-Faulting Memory Accesses) *Although our machine model encodes the cases for memory-access faults, within the context of this thesis (e.g.* inrf *in Listing 3.2), we only consider the accesses that are mapped to physical memory (e.g. the virtual address* w *mapped to the physical address* a *in Listing 3.2).*

```
(*Performs final level of address translation, starting from the PT entry
References: Intel Manual, Volume 3A - System Programming Guide Part 1
(September 2016) - pg. 4-23, PDF pg. 127 - Table 4-19 (pg. 4-27, PDF pg. 131)*)
```

```
Definition translate_from_pte (m: mem64) (pte: word 64) (w: word 64): word 64 + mem_fault :=
    check_fault_and_continue pte
        (fun _ ⇒ inl (fix_for_pte (create_entry_addr pte w 3 11 three_le11
                three_lt64 eleven_minus3plus1lt64minus3 phys_addr_is_64) w)).
(*Performs the fourth level of address translation, starting from the PD entry
References: Intel Manual, Volume 3A - System Programming Guide Part 1
(September 2016) - pg. 4-22, PDF pg. 126 - Table 4-18 (pg. 4-26, PDF pg. 130)*)
Definition translate_from_pde (m: mem64) (pde: word 64) (w: word 64): word 64 + mem_fault :=
    check_fault_and_continue pde
        (fun _ ⇒ translate_using_linear_addr_range m pde w 12 20
                twelve_le20 twelve_lt64 twenty_minus12plus1lt64minus12
                    pde_entry_addr_is_64 translate_from_pte).
(*Performs the third level of address translation, starting at the PDP table entry
References: Intel Manual, Volume 3A - System Programming Guide Part 1
(September 2016) - pg. 4-22, PDF pg. 126 - Table 4-16 (pg. 4-25, PDF pg. 129)*)
Definition translate_from_pdpte (m: mem64) (pdpte: word 64) (w: word 64): word 64 + mem_fault :=
    check_fault_and_continue pdpte
        (fun _ ⇒ translate_using_linear_addr_range m pdpte w 21 29
            twenty1_le29 twenty1_lt64 twenty9_minus21plus1lt64minus21
                pde_entry_addr_is_64 translate_from_pde).
(*Performs the second level of address translation, starting at the PML4 table entry.
References: Intel Manual, Volume 3A - System Programming Guide Part 1
(September 2016) - pg. 4-22, PDF pg. 126 - Table 4-14 (pg. 4-23, PDF pg. 127)*)
Definition translate_from_pml4e (m: mem64) (pml4e: word 64) (w: word 64): word 64 + mem_fault :=
    check_fault_and_continue pml4e
        (fun _ ⇒ translate_using_linear_addr_range m pml4e w 30 38
                thirty_le38 thirty_lt64 thirty8_minus30plus1lt64minus30
                    pdpt_entry_addr_is_64 translate_from_pdpte).
(*Performs the address translation, handling the first level `(to get the PML4 entry)
```

```
References: Intel Manual, Volume 3A - System Programming Guide Part 1
(September 2016) - pg. 4-22, PDF pg. 126 - Table 4-12 (pg. 4-19, PDF pg. 123)*)
Definition translate_top_level (m: mem64) (cr3val: word 64) (w: word 64): word 64 + mem_fault :=
    translate_using_linear_addr_range
        m cr3val w 39 47 thirty9_le47   thirty9_lt64
            forty7_minus39plus1lt64minus39 pml4_entry_addr_is_64
                translate_from_pml4e.
```

Listing 3.3 Translating a virtual address (w) to map a physical page, which is pictorially shown in Figure 3.1

The top-level call translate, when unfolded, includes a chain of calls per level of page tables to reach the physical page address. The first call is made for level 4 to obtain the physical address of the entry at the level, translate_top_level in Listing 3.3, with the physical address of the root of the page-table (cr3val) and the level 4 index residing in between the bits 39-47 from the virtual address (w). Then, the physical address of level 4 entry (plm4e) is passed to the second call of the chain (translate_from_plm4 in Listing 3.3) to obtain the physical address of level 3 entry via the level 3 index residing in between bits 30-38 of the virtual address (w). This chain of calls ends with the final level translation call (translate_from_pte) with the related indexing bits to the physical page address.

Load and Store From/To Memory The next two essential pieces, based on the address translation, are loading from and storing to memory as shown in Listing 3.4. Loading a value (val in Listing 3.4) from a virtual address (addr) first requires obtaining the physical address (phys_addr) via address-translation and loading the value using the physical address (P.load)

```
Definition load (m: virtmem64) (addr: address): word 64 + mem_fault :=
    match m with
      | virtmem phys root ⇒
        match translate phys root addr with
          | inr fault ⇒ inr fault
```

```
6           | inl phys_addr ⇒
7             match P.load (match m with | virtmem phys _ ⇒ phys end) phys_addr with
8               | inr fault ⇒ inr (phys_fault fault)
9               | inl val ⇒ inl val
10            end
11        end
12    end.
13(* This is a probe so if we jump/call unmapped memory, we fault immediately *)
14Definition code_probe (m: virtmem64) (addr: word 64): unit + mem_fault :=
15    match m with
16      | virtmem phys root ⇒
17        match translate_top_level phys (proj1_sig root) addr with
18          | inr fault ⇒ inr fault
19          | inl phys_addr ⇒
20            match P.probe phys phys_addr with
21              | inr fault ⇒ inr (phys_fault fault)
22              | inl val ⇒ inl tt
23            end
24        end
25    end.
26Definition store (m: virtmem64) (addr: address) (w: word 64): virtmem64 + mem_fault :=
27    match m with
28      | virtmem phys root ⇒
29        match translate phys root addr with
30          | inr fault ⇒ inr fault
31          | inl phys_addr ⇒
32            match P.store (match m with | virtmem phys _ ⇒ phys end) phys_addr w with
33              | (_, inr fault) ⇒ inr (phys_fault fault)
34              | (mem, _) ⇒ inl (virtmem mem root)
```

```
35            end
36          end
37      end.
```

Listing 3.4 Loading, Storing for Virtual Memory Addresses and Probing for Code Pointers

Likewise, storing a value (the argument w in store function in Listing 3.4) to a virtual address (addr) also first requires obtaining the physical memory address (phys_addr), then storing the value (P.store) using the physical address, and obtaining the updated memory (mem).

Queries over the existence of the virtual address mapping – *probing* ($\mathcal{Q}$) – provide the mechanism for ensuring the control-flow structure of the program constructed with instructions such as call, return, and jump is mapped in the memory.

3.4 Instructions

The last piece of our constructions to explain before defining operational semantics is *instructions*. In our model design, we index instructions with operation types: the constructors, which are in the type of InstructionType_ops (e.g., mov_ops), and the form of InstructionType_OperandType_OperandType (e.g., mov_reg64_reg64).

3.4.1 Handling Instructions Based on Operand Types

Essentially, most of the instructions do some operation over the values of source physical resources (e.g., a register or a memory address) and update the destination resource, where the updated value resides. Therefore, handling the updates after the operation over the resources in a more general form and clarity, we generalize the updates to the resources with

- identifying the operand types in grouping updates to the resources (e.g., reg_reg, reg_imm in Listing 3.5)
- parameterizing the need for *overflow* checks and flag updates (setflags in Listing 3.5)

```
Definition reg_reg (setflags: bool)
    (op: forall sz, WordImpl.word sz → WordImpl.word sz → WordImpl.word sz * bool)
    (regs: regset) (mem: VirtMemImpl.virtmem64) (dst: ireg) (src: ireg)
    : regset * VirtMemImpl.virtmem64 + fault_error :=
  let '(res,b) := (op 64 (regset_get_num regs (ir dst)) (regset_get_num regs (ir src))) in
  let regs := if setflags then apply_arith_flags regs
                (arith_flags.res_overflow_to_flags res b) else regs in
                    sret (regset_replace regs (ir dst) (num res), mem).
Definition reg_imm (setflags: bool)
    (op: forall sz, WordImpl.word sz → WordImpl.word sz → WordImpl.word sz * bool)
    (regs: regset) (mem: VirtMemImpl.virtmem64) (dst: ireg) (imm: WordImpl.word 64)
    : regset * VirtMemImpl.virtmem64 + fault_error :=
  let '(res,b) := (op 64 (regset_get_num regs (ir dst)) imm) in
  let regs := if setflags then apply_arith_flags regs
                (arith_flags.res_overflow_to_flags res b) else regs in
                    sret (regset_replace regs (ir dst) (num res), mem).
Definition reg_mem (setflags: bool)
    (op: forall sz, WordImpl.word sz → WordImpl.word sz → WordImpl.word sz * bool)
    (regs: regset) (mem: VirtMemImpl.virtmem64) (dst: ireg) (src: addressing_mode)
    : regset * VirtMemImpl.virtmem64 + fault_error :=
  do val ← memory_get_by_mode mem src regs;
  let '(res,b) := (op 64 (regset_get_num regs (ir dst)) val) in
  let regs := if setflags then apply_arith_flags regs
                (arith_flags.res_overflow_to_flags res b) else regs in
                    sret (regset_replace regs (ir dst) (num res), mem).
Definition mem_reg (setflags: bool)
    (op: forall sz, WordImpl.word sz → WordImpl.word sz → WordImpl.word sz * bool)
    (regs: regset) (mem: VirtMemImpl.virtmem64) (dst: addressing_mode) (src: ireg)
    : regset * VirtMemImpl.virtmem64 + fault_error :=
```

```
30   do val ← memory_get_by_mode mem dst regs;
31   let '(res,b) := (op 64 val (regset_get_num regs (ir src))) in
32   let regs := if setflags then apply_arith_flags regs
33                 (arith_flags.res_overflow_to_flags res b) else regs in
34                     do newmem ← memory_replace_by_mode mem dst res regs;
35                     sret (regs, newmem).
36 Definition mem_imm (setflags: bool)
37     (op: forall sz, WordImpl.word sz → WordImpl.word sz → WordImpl.word sz * bool)
38     (regs: regset) (mem: VirtMemImpl.virtmem64) (dst: addressing_mode) (imm: WordImpl.word 64)
39     : regset * VirtMemImpl.virtmem64 + fault_error :=
40   do val ← memory_get_by_mode mem dst regs;
41   let '(res,b) := (op 64 val imm) in
42   let regs := if setflags then apply_arith_flags regs
43                 (arith_flags.res_overflow_to_flags res b) else regs in
44                     do newmem ← memory_replace_by_mode mem dst res regs;
45                     sret (regs, newmem).
```

Listing 3.5 Handling Instructions with Register, Memory, and Immediate Value Arguments

Reg To Reg – RR Register to register operations (reg_reg) updates (regset_replace the destination register (ir dst) with the value obtained after the application of an operation (op) on the values of the source register's (ir src) (regset_get_num regs (ir src) and destination's (regset_get_num regs (ir dst).

Reg with Immediate Value – RImm Likewise, a destination register can be updated with an immediate value.

Reg From Memory – RM While accessing a memory, we could do it by an offset to an address or an absolute one (textsfaddress_mode). After reading a value (memory_get_by_mode) of a source memory address (kept in src register), we can apply the operation over the current value of memory value (memory_get_by_mode mem src regs) and a destination register's one value, then update the

destination register's value.

Memory from Reg – MR Likewise, we can also update the value of a memory address (mem_replace_by_mode) with a new value obtained by applying an operation over a source register value (regset_get_num regs (ir src)) and the current value of the memory address.

3.5 Giving Semantics to Instructions

In this section, with the well-enough definitions we have so far, we can give operational semantics to the instructions per operation type. We should emphasize that the instructions used in this thesis do not reflect any change on the CPU component of the state ($\sigma.\mathcal{C}$), and since our principles cannot handle the failure (Remark 4), we ignore the faulting cases of the instructions result (instr_result and arith_result in Listing 3.6. To use the general form (presented in Listing 3.5) for handling the effects of instructions, we present a bridge (arith_instr) which asks for exactly what RR, RM, MR, and RImm in Listing 3.5 need per operand type.

Notably on Listing 3.6, we should mention that, operationally, updating control (ctlarg in Figure 3.6) registers is not different than updating a general purpose register, but it is logically different, and we discuss it in our reasoning principles. The individual instructions identify the change in the instruction pointer (rip) as the size of instructions varies and the kind of flags to be set if asked, i.e., setflag is set.

```
1 Inductive arith_ops: Type :=
2   (* r/m64, imm32 *)
3   | arith_reg64_imm32: ireg → WordImpl.word 32 → arith_ops
4   | arith_mem64_imm32: addressing_mode → WordImpl.word 32 → arith_ops
5   (* r/m64, r64 *)
6   | arith_reg64_reg64: ireg → ireg → arith_ops
7   | arith_mem64_reg64: addressing_mode → ireg → arith_ops
8   (* r64, r/m64 *)
9   | arith_reg64_mem64: ireg → addressing_mode → arith_ops.
```

```
Definition arith_instr_new
    (op: forall sz, WordImpl.word sz → WordImpl.word sz → WordImpl.word sz * bool)
    (cpu: cpu_int) (regs: regset) (mem: VirtMemImpl.virtmem64) (ops: arith_ops)
    : arith_result :=
  let (dstop, srcop) :=
      match ops with
        | arith_reg64_imm32 dst src ⇒
                        (regarg dst, immarg (WordImpl.concat (WordImpl.zero 32) src))
        | arith_mem64_imm32 dst src ⇒
                        (memarg dst, immarg (WordImpl.concat (WordImpl.zero 32) src))
        | arith_reg64_reg64 dst src ⇒ (regarg dst, regarg src)
        | arith_mem64_reg64 dst src ⇒ (memarg dst, regarg src)
        | arith_reg64_mem64 dst src ⇒ (regarg dst, memarg src)
      end in
    arith_instr true op cpu regs mem dstop srcop.
Definition instr_result: Type := (cpu_int * regset * VirtMemImpl.virtmem64 + fault_error).
Definition arith_result: Type := (cpu_int * regset * VirtMemImpl.virtmem64 + fault_error)
Definition arith_instr (setflags: bool)
    (op: forall sz, WordImpl.word sz → WordImpl.word sz → WordImpl.word sz * bool)
    (cpu: cpu_int) (regs: regset) (mem: VirtMemImpl.virtmem64)
    (dst: instrarg) (src: instrarg)
    : arith_result :=
  do result ←
      match dst, src with
        | (regarg dstreg), (regarg srcreg) ⇒ reg_reg setflags op regs mem dstreg srcreg
        | (regarg dstreg), (immarg immval) ⇒ reg_imm setflags op regs mem dstreg immval
        | (regarg dstreg), (memarg srcmem) ⇒ reg_mem setflags op regs mem dstreg srcmem
        | (memarg dstmem), (regarg srcreg) ⇒ mem_reg setflags op regs mem dstmem srcreg
        | (memarg dstmem), (immarg immval) ⇒ mem_imm setflags op regs mem dstmem immval
```

```
39        | (ctlarg dstctl), (regarg regval) ⇒ ctl_reg regs mem dstctl regval
40        | _,                _                ⇒ inr (UD, None)
41      end;
42  let '(newregs, newmem) := result in
43    sret (cpu, newregs, newmem).
```

Listing 3.6 Instruction Result

3.5.1 Reading To/From Virtual Memory Address

In Figure 3.7, we see mov_instr instruction including operand type resolution (Lines 8 - 14) to identify the assignment function shown in Listing 3.5. Our bridge to the arith_instr is move_arith in Listing 3.7 which does not ask for any check on setflag and performs non_overflowing operations – i.e., non-arithmetic and just assignment operation.

```
1  Definition move_arith := arith_instr false (non_overflowing wsecond).
2  Definition mov_instr
3     (cpu: cpu_int) (regs: regset)
4     (mem: VirtMemImpl.virtmem64) (ops: mov_ops)
5     : instr_result :=
6    let (dstop, srcop) :=
7       match ops with
8         | mov_reg64_reg64 dst src ⇒ (regarg dst, regarg src)
9         | mov_mem64_reg64 dst src ⇒ (memarg dst, regarg src)
10        | mov_reg64_mem64 dst src ⇒ (regarg dst, memarg src)
11        | mov_reg64_imm64 dst src ⇒ (regarg dst, immarg src)
12        | mov_reg64_imm32 dst src ⇒ (regarg dst,
13                                      immarg (WordImpl.concat (WordImpl.zero 32) src))
14        | mov_mem64_imm32 dst src ⇒ (memarg dst,
15                                      immarg (WordImpl.concat (WordImpl.zero 32) src))
16      end in
```

```
17    match dstop with
18      (*| selarg cs ⇒ inr (UD, None) Can't mov to code segment register *)
19        | _ ⇒
20        do result ← move_arith cpu regs mem dstop srcop;
21        let '(newcpu, newregs, newmem) := result in
22          sret (newcpu, regset_add_rip_nat newregs (mov_ops_to_length ops), newmem)
23    end.
```

Listing 3.7 mov Instructions

StepMovRR

$$\frac{\begin{array}{c} ip + \mathsf{MovLenRR} = ip' \qquad \sigma.\mathcal{R}[rip] = ip \\ \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \qquad \sigma.\mathcal{R}[r_{\mathsf{src}}] = v_s \qquad \mathsf{mov\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{reg64\_reg64}(r_{\mathsf{dst}}, r_{\mathsf{src}})) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \\ \sigma'.\mathcal{R}[r_{\mathsf{dst}} = v_s \qquad \sigma'.\mathcal{R}[r_{\mathsf{src}}] = v_s \qquad \sigma'.\mathcal{R}[rip] = ip' \end{array}}{\mathsf{mov}\ r_{\mathsf{dst}}\ r_{\mathsf{src}} \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepMovRImm

$$\frac{\begin{array}{c} ip + \mathsf{MovLenRM} = ip' \qquad \sigma.\mathcal{R}[rip] = ip \\ \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \qquad \mathsf{mov\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{mov\_reg64\_imm64}(r_{\mathsf{dst}}, v)) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \\ \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v \qquad \sigma'.\mathcal{R}[rip] = ip' \end{array}}{\mathsf{mov}\ r_{\mathsf{dst}}\ v \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepMovRMBase

$$\frac{\begin{array}{c} \mathsf{addressing\_mode} = \mathsf{base} \qquad ip + \mathsf{MovLenRMBase} = ip' \qquad \sigma.\mathcal{R}[rip] = ip \qquad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \\ \sigma.\mathcal{M}[src] = v \qquad \mathsf{mov\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{mov\_reg64\_mem64}(r_{\mathsf{dst}}, src)) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \\ \sigma'.\mathcal{R}[r_{\mathsf{dst}} = v] \qquad \sigma'.\mathcal{R}[rip] = ip' \end{array}}{\mathsf{mov}\ r_{\mathsf{dst}}\ src \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepMovRMOff

$$\frac{\begin{array}{c} \mathsf{addressing\_mode} = \mathsf{offset} \qquad ip + \mathsf{MovLenRMOff} = ip' \qquad \sigma.\mathcal{R}[rip] = ip \qquad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \\ \sigma.\mathcal{M}[src + offset] = v \qquad \mathsf{mov\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{mov\_reg64\_mem64}(r_{\mathsf{dst}}, src)) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \\ \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v \qquad \sigma'.\mathcal{R}[rip] = ip' \end{array}}{\mathsf{mov}\ r_{\mathsf{dst}}\ src \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepMovMRBase

$$\frac{\begin{array}{c} \mathsf{addressing\_mode} = \mathsf{base} \qquad ip + \mathsf{MovLenMRBase} = ip' \qquad \sigma.\mathcal{R}[rip] = ip \qquad \sigma.\mathcal{R}[r_{\mathsf{src}}] = v_s \\ \sigma.\mathcal{M}[dst] = v \qquad \mathsf{mov\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{mov\_mem64\_reg64}(dst, r_{\mathsf{src}})) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma'.\mathcal{M}) \\ \sigma'.\mathcal{M}[dst] = v_s \qquad \sigma'.\mathcal{R}[rip] = ip' \end{array}}{\mathsf{mov}\ dst\ r_{\mathsf{src}} \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepMovMROff

$$\frac{\begin{array}{c} \mathsf{addressing\_mode} = \mathsf{offset} \qquad ip + \mathsf{MovLenMROffset} = ip' \qquad \sigma.\mathcal{R}[rip] = ip \qquad \sigma.\mathcal{R}[r_{\mathsf{src}}] = v_s \\ \sigma.\mathcal{M}[dst + offset] = v \qquad \mathsf{mov\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{mov\_mem64\_reg64}(dst, r_{\mathsf{src}})) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \\ \sigma'.\mathcal{M}[dst + offset] = v_s \qquad \sigma'.\mathcal{R}[rip] = ip' \end{array}}{\mathsf{mov}\ dst\ r_{\mathsf{src}} \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

Figure 3.5 Operational Rules for Selected mov Instructions

Reduction Rules for mov Instructions In Figure 3.5, we see the reduction rules for each of the mov operations types into which mov_instr resolve in Listing 3.7. In each of these rules, we see that instruction length varies (e.g., MovLenMRBase) and the instruction pointer (rip) is set accordingly – regset_add_rip_nat newregs (mov_ops_to_length ops) in Listing 3.7. Based on the operand type, updates occur either on a register (r_{dst}) or memory location (dst) from a memory (src) or register (r_{src}) source. Regarding the rules with memory accesses, we see separate rules based on the *access-mode*, i.e., base or offset.

3.5.2 Arithmetic Operations

3.5.3 Add, Sub and Compare Instructions

In Listing 3.8, we see add_instr instruction, which is constructed with a constructor arith_instr_new (Line 10 in Listing 3.6) that takes wplus_check as an operation, which sums the operand values with an overflow check, and as we see in the constructor, setflags is set, i.e., the values of flags are to be resolved. Likewise, for sub_instr instruction, which is also constructed with arith_instr_new, but, for substitution, it takes wminus_check as an operation and setflag is set to have the flags to be resolved.

```
1  Definition add_instr (cpu: cpu_int) (regs: regset)
2                       (mem: VirtMemImpl.virtmem64) (ops: arith_ops) :=
3    do result ← arith_instr_new wplus_check cpu regs mem ops;
4    let '(newcpu, newregs, newmem) := result in
5      sret (newcpu, regset_add_rip_nat newregs (add_ops_to_length ops), newmem).
6  Definition cmp_instr (cpu: cpu_int) (regs: regset)
7                       (mem: VirtMemImpl.virtmem64) (ops: arith_ops) :=
8    do result ← arith_instr_new wminus_check cpu regs mem ops;
9    let '(newcpu, subregs, submem) := result in
10   (* Notice that we return the old memory on purpose! *)
11   (* We copy the flags after sub into the final regs,
12     in order to get the right flag updates from sub *)
13   let correctregs := regset_replace regs (fl rflags) (regset_get subregs (fl rflags)) in
```

```
14  sret (newcpu, regset_add_rip_nat correctregs (cmp_ops_to_length ops), mem).
```

Listing 3.8 Arithmetic Operations

Regarding the comparison, we model the instruction cmp_instr using the sub_instr in a slightly different way to reflect the changes in the flags inside the register set (correctregs in Listing 3.8) while keeping the memory intact as shown in Listing 3.8 – (newcpu, regset_add_rip_nat correctregs (cmp_ops_to_length ops), mem).

Remark 2 (Arithmetic without Memory Access) *Although we model all the memory-accessing versions of the arithmetic instructions, within the context of this thesis, we use and consider explaining only the non-memory-accessing versions. This makes all the arithmetic instructions (*add, sub *and* cmp*) keep the memory intact.*

Reduction Rules for add, sub, and cmp In Figure 3.6, we see the reduction rules for subtraction, addition, and comparison. Since setflag for these instructions is set, we see the change in the fl rflags register. The flags we consider for the arithmetic instructions are ZF, SF, OF and PF, and we can observe the change in them via apply_arith_flags: overflow check for sume is arith_flags.res_overflow_to_flags $(v_s + v_d, b)$.

3.5.4 Shift, And, Or, and Xor Instructions

Unlike addition, subtraction and comparison, the bitwise instructions are run with non-overflow-check, e.g., non_overflowing wand. However, executing them still requires access to the flag register.

```
1  Definition shr_instr (cpu: cpu_int) (regs: regset)
2                       (mem: VirtMemImpl.virtmem64)
3                       (ops: shift_ops) :=
4    do result ← shift_instr wshiftrl cpu regs mem ops;
5    let '(newcpu, newregs, newmem) := result in
6      sret (newcpu, regset_add_rip_nat newregs (shr_ops_to_length ops), newmem).
```

StepAddRR

$$\frac{\begin{array}{c} ip + \mathsf{AddLenRR} = ip' \qquad \sigma.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags} \qquad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \qquad \sigma.\mathcal{R}[rip] = ip \\ \sigma.\mathcal{R}[r_{\mathsf{src}}] = v_s \qquad \mathsf{apply\_arith\_flags}(\sigma.\mathcal{R}, \mathsf{arith\_flags.res\_overflow\_to\_flags}(v_s + v_d, b))[\mathsf{rflags}] = \mathsf{flags}' \\ \mathsf{add\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_reg64}(r_{\mathsf{dst}}, r_{\mathsf{src}})) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \\ \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v_s + v_d \qquad \sigma'.\mathcal{R}[r_{\mathsf{src}}] = v_s \qquad \sigma'.\mathcal{R}[rip] = ip' \qquad \sigma'.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags}' \end{array}}{\mathsf{add}\ r_{\mathsf{dst}}\ r_{\mathsf{src}} \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepAddRImm

$$\frac{\begin{array}{c} ip + \mathsf{AddLenRImm} = ip' \qquad \sigma.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags} \qquad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \\ \sigma.\mathcal{R}[rip] = ip \qquad \mathsf{apply\_arith\_flags}(\sigma.\mathcal{R}, \mathsf{arith\_flags.res\_overflow\_to\_flags}(v_s + v_d, b))[\mathsf{rflags}] = \mathsf{flags}' \\ \mathsf{add\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_imm32}(r_{\mathsf{dst}}, v)) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \\ \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v + v_d \qquad \sigma'.\mathcal{R}[rip] = ip' \qquad \sigma'.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags}' \end{array}}{\mathsf{add}\ r_{\mathsf{dst}}\ v \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepSubRR

$$\frac{\begin{array}{c} ip + \mathsf{SubLenRR} = ip' \qquad \sigma.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags} \qquad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \qquad \sigma.\mathcal{R}[rip] = ip \\ \sigma.\mathcal{R}[r_{\mathsf{src}}] = v_s \qquad \mathsf{apply\_arith\_flags}(\sigma.\mathcal{R}, \mathsf{arith\_flags.res\_overflow\_to\_flags}(v_s + v_d, b))[\mathsf{rflags}] = \mathsf{flags}' \\ \mathsf{sub\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_reg64}(r_{\mathsf{dst}}, r_{\mathsf{src}})) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \\ \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v_s - v_d \qquad \sigma'.\mathcal{R}[r_{\mathsf{src}}] = v_s \qquad \sigma'.\mathcal{R}[rip] = ip' \qquad \sigma'.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags}' \end{array}}{\mathsf{sub}\ r_{\mathsf{dst}}\ r_{\mathsf{src}} \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepSubRImm

$$\frac{\begin{array}{c} ip + \mathsf{SubLenRImm} = ip' \qquad \sigma.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags} \qquad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \\ \sigma.\mathcal{R}[rip] = ip \qquad \mathsf{apply\_arith\_flags}(\sigma.\mathcal{R}, \mathsf{arith\_flags.res\_overflow\_to\_flags}(v_s + v_d, b))[\mathsf{rflags}] = \mathsf{flags}' \\ \mathsf{sub\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_imm32}(r_{\mathsf{dst}}, v)) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \\ \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v_d - v \qquad \sigma'.\mathcal{R}[rip] = ip' \qquad \sigma'.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags}' \end{array}}{\mathsf{sub}\ r_{\mathsf{dst}}\ v \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepCmpRR

$$\frac{\begin{array}{c} ip + \mathsf{CmpLenRR} = ip' \qquad \sigma.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags} \qquad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \qquad \sigma.\mathcal{R}[rip] = ip \\ \sigma.\mathcal{R}[r_{\mathsf{src}}] = v_s \qquad \mathsf{apply\_arith\_flags}(\sigma.\mathcal{R}, \mathsf{arith\_flags.res\_overflow\_to\_flags}(v_s + v_d, b))[\mathsf{rflags}] = \mathsf{flags}' \\ \mathsf{cmp\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_reg64}(r_{\mathsf{dst}}, r_{\mathsf{src}})) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \\ \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v_s - v_d \qquad \sigma'.\mathcal{R}[r_{\mathsf{src}}] = v_s \qquad \sigma'.\mathcal{R}[rip] = ip' \qquad \sigma'.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags}' \end{array}}{\mathsf{cmp}\ r_{\mathsf{dst}}\ r_{\mathsf{src}} \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

Figure 3.6 Operational Rules for add, sub and cmp Instructions

Listing 3.9 Shift Left/Right Instructions

3.5.5 Stack Operations

Stack operations enable manipulating the stack pointer register rsp to be constitute local context for execution. As we see in Listing 3.10, pushing a value (val) onto stack simply decrements the register of the current stack pointer (rsp) by 8 bytes and writes the value into that 8 bytes decremented memory address (memresult). On the other hand, when popping a value from the stack, the memory

StepAndRR

$$\frac{ip + \mathsf{AndLenRR} = ip' \quad \sigma.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags} \quad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \quad \sigma.\mathcal{R}[rip] = ip \quad \sigma.\mathcal{R}[r_{\mathsf{src}}] = v_s \quad \mathsf{and\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_reg64}(r_{\mathsf{dst}}, r_{\mathsf{src}})) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \quad \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v_s \& v_d \quad \sigma'.\mathcal{R}[r_{\mathsf{src}}] = v_s \quad \sigma'.\mathcal{R}[rip] = ip' \quad \sigma'.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags}}{\mathsf{add}\ r_{\mathsf{dst}}\ r_{\mathsf{src}} \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepAndRImm

$$\frac{ip + \mathsf{AndLenRImm} = ip' \quad \sigma.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags} \quad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \quad \sigma.\mathcal{R}[rip] = ip \quad \mathsf{and\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_imm32}(r_{\mathsf{dst}}, v)) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \quad \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v \& v_d \quad \sigma'.\mathcal{R}[rip] = ip' \quad \sigma'.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags}}{\mathsf{and}\ r_{\mathsf{dst}}\ v \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepOrRR

$$\frac{ip + \mathsf{OrLenRR} = ip' \quad \sigma.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags} \quad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \quad \sigma.\mathcal{R}[rip] = ip \quad \sigma.\mathcal{R}[r_{\mathsf{src}}] = v_s \quad \mathsf{or\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_reg64}(r_{\mathsf{dst}}, r_{\mathsf{src}})) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \quad \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v_s || v_d \quad \sigma'.\mathcal{R}[r_{\mathsf{src}}] = v_s \quad \sigma'.\mathcal{R}[rip] = ip' \quad \sigma'.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags}}{\mathsf{or}\ r_{\mathsf{dst}}\ r_{\mathsf{src}} \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepOrRImm

$$\frac{ip + \mathsf{OrLenRImm} = ip' \quad \sigma.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags} \quad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \quad \sigma.\mathcal{R}[rip] = ip \quad \mathsf{or\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_imm32}(r_{\mathsf{dst}}, v)) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \quad \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v_d || v \quad \sigma'.\mathcal{R}[rip] = ip' \quad \sigma'.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags}}{\mathsf{or}\ r_{\mathsf{dst}}\ v \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepShlRImm

$$\frac{ip + \mathsf{ShlLenRImm} = ip' \quad \sigma.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags} \quad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \quad \sigma.\mathcal{R}[rip] = ip \quad (\mathsf{arith\_flags.res\_overflow\_to\_flags}(\sigma.\mathcal{R}, v_d << v, b)[\mathsf{rflags}] = \mathsf{flags}' \quad \mathsf{shl\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_imm32}(r_{\mathsf{dst}}, v)) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \quad \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v_d << v \quad \sigma'.\mathcal{R}[rip] = ip' \quad \sigma'.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags}}{\mathsf{shl}\ r_{\mathsf{dst}}\ v \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

StepShrRImm

$$\frac{ip + \mathsf{ShrLenRImm} = ip' \quad \sigma.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags} \quad \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \quad \sigma.\mathcal{R}[rip] = ip \quad \mathsf{shr\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_imm32}(r_{\mathsf{dst}}, v)) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \quad \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v_d >> v \quad \sigma'.\mathcal{R}[rip] = ip' \quad \sigma'.\mathcal{R}[\mathsf{rflags}] = \mathsf{flags}}{\mathsf{shr}\ r_{\mathsf{dst}}\ v \ /\ \sigma \longrightarrow \mathsf{skip} \ /\ \sigma'}$$

Figure 3.7 Selected Operational Rules for Bitwise Instructions

address kept in the register of the stack pointer is read, and the value kept in the register of the stack pointer is incremented by 8 bytes.

```
1 Definition memory_stack_push
2 (mem: VirtMemImpl.virtmem64) (regs: regset) (val: WordImpl.word 64)
3    : VirtMemImpl.virtmem64 * regset + fault_error :=
4  let rspval := regset_get_num regs (ir rsp) in
5  let newrspval := VirtMemImpl.address_minus (exist _ rspval I) w8_64_aligned in
6  let memresult := memory_replace mem newrspval val in
```

```
  let newregs := regset_replace regs (ir rsp) (num (proj1_sig newrspval)) in
    do newmem ← memresult; sret (newmem, newregs).
Definition memory_stack_pop (mem: VirtMemImpl.virtmem64) (regs: regset)
    : WordImpl.word 64 * regset + fault_error :=
  let rspval := (exist _ (regset_get_num regs (ir rsp)) I) in
  let newrspval := VirtMemImpl.address_plus rspval w8_64_aligned in
  let memresult := memory_get mem rspval in
  let newregs := regset_replace regs (ir rsp) (num (proj1_sig newrspval)) in
    do val ← memresult; sret (val, newregs).
```

Listing 3.10 Push and Pop To/From Memory

Reduction Rules for Stack Operations In Figure 3.8, we give the reduction rules for pushing a register value (v_s) onto the stack (StackPushR), and, reversly, reading the stack value (v_s) into a register (StepPopR).

$$\textsc{StepPushR}\quad \frac{ip + \mathsf{PushLenRR} = ip' \qquad \sigma.\mathcal{R}[\mathsf{rsp}] = \mathsf{maddr} \qquad \sigma.\mathcal{R}[rip] = ip \qquad \sigma.\mathcal{R}[r_{\mathsf{src}}] = v_s \qquad \sigma.\mathcal{M}[\mathsf{maddr} - 8] = v \qquad \mathsf{push\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_reg64}(r_{\mathsf{dst}}, r_{\mathsf{src}})) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma'.\mathcal{M}) \qquad \sigma'.\mathcal{R}[\mathsf{rsp}] = \mathsf{maddr} - 8 \qquad \sigma'.\mathcal{R}[r_{\mathsf{src}}] = v_s \qquad \sigma'.\mathcal{R}[rip] = ip' \qquad \sigma'.\mathcal{M}[\mathsf{maddr} - 8] = v_s}{\mathsf{push}\ r_{\mathsf{src}}\ /\ \sigma \longrightarrow \mathsf{skip}\ /\ \sigma'}$$

$$\textsc{StepPopR}\quad \frac{ip + \mathsf{PopLenRR} = ip' \qquad \sigma.\mathcal{R}[\mathsf{rsp}] = \mathsf{maddr} \qquad \sigma.\mathcal{R}[rip] = ip \qquad \sigma.\mathcal{R}[r_{\mathsf{src}}] = v_s \qquad \sigma.\mathcal{M}[\mathsf{maddr}] = v \qquad \mathsf{push\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{arith\_reg64\_reg64}(r_{\mathsf{dst}}, r_{\mathsf{src}})) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma'.\mathcal{M}) \qquad \sigma'.\mathcal{R}[\mathsf{rsp}] = \mathsf{maddr} + 8 \qquad \sigma'.\mathcal{R}[r_{\mathsf{src}}] = v \qquad \sigma'.\mathcal{R}[rip] = ip' \qquad \sigma'.\mathcal{M}[\mathsf{maddr} - 8] = v}{\mathsf{push}\ r_{\mathsf{src}}\ /\ \sigma \longrightarrow \mathsf{skip}\ /\ \sigma'}$$

Figure 3.8 Selected Operational Rules for Pop and Push Instructions

3.5.6 Control-Flow Instructions

Control-flow operations change the flow of instruction execution. While doing so, the essential aspect that the model guarantees is having the instructions inside the changed flow, which themselves live in the memory, mapped in the memory. Up until now, the semantics of the instructions explained so

far heavily rely on load and store from Listing 3.4, unlike the control-flow instructions, which use code_probe for checking whether *code-to-be-executed* is mapped or not.

Jump To an Address In Listing 3.11, we see the jump instruction implementation, which jumps to an address from the current instruction—where the current rip shows—to an instruction with an offset (w). To do so, memory probing—checking whether the instruction at the target address (nextrip) is done. The existence of non-failure probing establishes the validity of the target to be jumped.

```
Program Definition jmp_instr (cpu: cpu_int) (regs: regset)
                             (mem: VirtMemImpl.virtmem64)
                             (ops: jmp_ops): instr_result :=
  match ops with
  | jrel32 w ⇒ let offset := sign_extend_to w 64 le_32_64 le_1_32 in
               let nextrip := WordImpl.wplus (regset_get_num regs (ir rip)) offset in
               do _ ← memory_probe mem nextrip;
               sret (cpu, regset_add_rip regs offset, mem)
    ...
  end.
```

Listing 3.11 Jump Instruction

$$\textsc{StepJmp}\quad \frac{\begin{array}{c}\sigma.\mathcal{R}[rip] = ip \qquad \sigma.\mathcal{R}[flrflags] = \mathsf{flagl} \\ \mathsf{ZFcond} = \mathsf{flagset\_includes\ flagl\ ZF} \qquad \mathcal{Q}(ip') = () \qquad \mathcal{C}\wr\lceil\rceil\mathcal{A}\sqcup\mathcal{C}\wr\backslash\lceil(\mathsf{ip}, \mathsf{offset}) = (\mathsf{is}, \mathsf{es}, ZFcond) \\ \mathsf{jmp\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{jrel32}(offset)) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \qquad \sigma'.\mathcal{R}[flrflags] = \mathsf{flagl}\end{array}}{\mathsf{jmpE\ offset}\ /\ \sigma \longrightarrow \mathsf{is}\ /\ \sigma'}$$

Figure 3.9 A Selected Operational Rule for Jump Instruction

Reduction Rule for a Jump Instruction In Figure 3.9, we see a rule for a jump instruction with a target address computable with an offset (offset). The next instruction pointer ($\sigma.\mathcal{R}[rip]$) and the next instruction sequence (the value of ip') are picked up on the condition (ZFcond). No

matter which target is picked up based on the condition, the rule has to make sure the non-failure code probing in the target address ($\mathcal{Q}(ip') = ()$), and the sequence of instructions encoded in byte sequences in the target address ($\mathcal{C}\wr\dashv\lceil\rceil\mathcal{A}\sqcup(\mathsf{ip'}) = (\mathsf{is}, \mathsf{es})$).

Call and Return To/From a Function The other two control-flow instructions are call and return. The code probing for callind in Listing 3.12 is done for the target—the value (target) held in the target register (r). Before doing that, the address of the next instruction after call is saved on the stack (retaddr). Conversely, ret_instr pops the return address value from the stack and then does code probing on that address. These two instructions—relying on the stack instructions—form consistent local execution contexts.

```
Definition call_instr
    (cpu: cpu_int) (regs: regset)
    (mem: VirtMemImpl.virtmem64) (ops: call_ops): instr_result :=
  let target := match ops with
    | callind r ⇒ regset_get_num regs (ir r)
    ...
    end in
        let retaddr := wplus (regset_get_num regs (ir rip))
                             (WordImpl.from_nat 64
                             (call_ops_length ops)) in
            do push_ret ← memory_stack_push mem regs retaddr;
            let '(mem_with_retaddr, regs_with_rsp_update) :=
                push_ret in
                    do _ ← memory_probe mem target;
                        sret (cpu, regset_set_rip regs_with_rsp_update
                                target, mem_with_retaddr).
Definition ret_instr (cpu: cpu_int) (regs: regset)
                     (mem: VirtMemImpl.virtmem64): instr_result :=
    do pop1 ← memory_stack_pop mem regs;
```

```
20      let (rip_val, regs_with_new_rsp) := pop1 in
21          do _ ← memory_probe mem rip_val;
22          sret (cpu, regset_set_rip regs_with_new_rsp rip_val, mem).
```

Listing 3.12 Call and Return Instructions

StepCall

$$\frac{\begin{array}{c} ip + \mathsf{CallLen} = ip' \qquad \mathcal{Q}(v_d) = () \\ \mathsf{CodeAt}(v_d) = (\mathsf{is}, \mathsf{es}) \qquad \mathsf{CodeAt}(ip') = (\mathsf{isret}, \mathsf{esret}) \qquad \sigma.\mathcal{M}.[maddr - 8] = mv \qquad \sigma.\mathcal{R}.[\mathsf{rsp}] = \mathsf{maddr} \\ \sigma.\mathcal{R}[r_{\mathsf{dst}}] = v_d \qquad \sigma.\mathcal{R}[rip] = ip \qquad \mathsf{call\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{callind}(\mathsf{off})) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \\ \sigma'.\mathcal{R}[r_{\mathsf{dst}}] = v_d \qquad \sigma'.\mathcal{R}.[\mathsf{rsp}] = \mathsf{maddr} \qquad \sigma'.\mathcal{R}[rip] = ip' \end{array}}{\mathsf{call}\ r_{\mathsf{dst}} \ /\ \sigma \longrightarrow \mathsf{is} \ /\ \sigma'}$$

StepReturn

$$\frac{\begin{array}{c} ip + \mathsf{ReturnLen} = ip' \qquad \mathcal{Q}(ip') = () \qquad \mathsf{CodeAt}(ip') = (\mathsf{is}, \mathsf{es}) \sigma.\mathcal{M}.[maddr] = mv \\ \sigma.\mathcal{R}.[\mathsf{rsp}] = \mathsf{maddr} \qquad \sigma.\mathcal{R}[rip] = ip \qquad \mathsf{call\_instr}(\sigma.\mathcal{C}, \sigma.\mathcal{R}, \sigma.\mathcal{M}, \mathsf{callind}(\mathsf{off})) = (\sigma.\mathcal{C}, \sigma'.\mathcal{R}, \sigma.\mathcal{M}) \\ \sigma'.\mathcal{M}.[maddr] = mv \qquad \sigma'.\mathcal{R}.[\mathsf{rsp}] = \mathsf{maddr} + 8 \qquad \sigma'.\mathcal{R}[rip] = mv \end{array}}{\mathsf{ret}\ /\ \sigma \longrightarrow \mathsf{is} \ /\ \sigma'}$$

Figure 3.10 Selected Operational Rules for Call and Return Instructions

Reduction Rules for Call and Return Instructions In addition to the code probing for the target ($\mathcal{Q}(v_d)$), the rule StepCall in Listing 3.10 also requires the association of the instruction sequences and correspoinding byte encoding of these sequences for both the jumped target ($\mathsf{CodeAt}(v_d) = (\mathsf{is},\mathsf{es})$) and the execution after the return ($\mathsf{CodeAt}(ip') = (\mathsf{isret},\mathsf{esret})$).

CHAPTER 4

PROGRAM LOGIC

We describe a program logic (a separation logic) along the lines suggested earlier, where every assertion is relative to an address space in which it is interpreted, allowing us to define *virtual points-to* assertions that make claims about memory locations in a particular address space. Virtual addresses, and even virtual points-to assertions, are not tagged with their address spaces in any way. Memory access in this logic is validated through the use of virtual points-to assertions in pre-conditions, which guarantee that address translations succeed. This supports rules for updating not only typical data in memory that happens to be subject to address translation, but manipulation of the page tables themselves via virtual addresses (as demanded by all modern hardware), also via virtual points-to assertions. To support specifications that deal with multiple address spaces, our logic incorporates a hybrid-style modality $[r](P)$ to state that an assertion is true in another (assertion-specified) address space rather than the address space currently active in hardware, which is not only useful for virtual memory manager invariants but critical to reasoning about change-of-address-space. By developing this within the Iris framework, we obtain additional features (e.g., fractional permissions) that allow us to verify some of the most subtle and technically challenging instruction sequences in an OS kernel (Section 5).

To support making assertions depend on a choice of address space, we work entirely in a pointwise lifting of Iris's base BI logic, essentially working with separation logic assertions indexed by a choice of page table root as a $\mathcal{W}_{64}$, which we call $\mathsf{vProp}\ \Sigma$:[1]

[1] Iris experts may notice our $-\!\!\mathrm{b}\!>$ resembles another variety of pointwise lifting that already exists in the main Iris distribution[41,42]. This is essentially true, but the standard formalization does not work for index types which are indexed in the sense of Coq's metatheory, as our word n types are.

```
Definition vProp Σ : bi := word 64 —b> iPropI Σ.
```

This is the (Coq) type of assertions in our logic. Most constructs in Iris's base logic are defined with respect to any BI-algebra (of Coq type bi), so automatically carry over to our derived logic. However, we must still build up from existing Iris primitives to provide new primitives that depend on the address space — primarily the notion of virtual points-to.

4.1 Base Points-To Assertions

We build up our context-dependent assertions from two basic types of assertions that do not care about address spaces.

4.1.1 Register points-to

The assertion $\mathsf{r} \mapsto_{\mathsf{r}} \{q\}\ \mathsf{rv}$ ensures the ownership of the register r naming the register value rv. The fraction q with value 1 asserts the unique ownership of the register mapping, and grants update permission on it, otherwise, any value $0 < \mathsf{q} < 1$ represents partial ownership granting readonly permission on the mapping.

4.1.2 Physical memory points-to

The soundness proofs for our logic's rules largely center around proving that page-table-walk accesses as in Figure 3.1 succeed, which requires assertions dealing with physical memory locations. We have two notions of physical points-to facts. The primitive notion closest to our machine model is captured by an assertion $\mathsf{pfn} \sim \mathsf{pageoff} \mapsto_{\mathsf{a}} \{\mathsf{q}\}\ \mathsf{v}$, where pfn (a $\mathcal{W}_{52}$ *page frame number*) essentially selects a 4KB page of physical memory, and pageoff (a $\mathcal{W}_{12}$) is an offset within that page. From this we can derive a more concise physical points-to when the split is unimportant:

$$\mathsf{w} \mapsto_{\mathsf{p}} \{q\}\ \mathsf{v} \triangleq (\mathsf{drop}\ 12\ w) \sim (\mathsf{bottom}\ 12\ w) \mapsto_{\mathsf{a}} \{\mathsf{q}\}\ \mathsf{v}$$

```
Definition va ↦t {q} v : vProp Σ :=
∃l4e l3e l2e l1e. ⌜ aligned va⌝∗ L4_L1_PointsTo(va l4e l3e l2e l1e paddr) ∗ paddr ↦p {q}v.

Definition L4_L1_PointsTo (maddr l4e l3e l2e l1e paddr :word 64) : vProp Σ := λ cr3val.
 ⌜ entry_present l4e ∧ entry_present l3e ∧ entry_present l2e ∧ entry_present l1e⌝ ∗
 (l4M52 maddr cr3val) ∼ (l4off maddr cr3val) ↦a {q1} l4e ∗
 (l3M52 maddr l4e) ∼ (l3off maddr l4e) ↦a {q2} l3e ∗
 (l2M52 maddr l3e) ∼ (l2off maddr l3e) ↦a {q3} l2e ∗
 (l1M52 maddr l2e) ∼ (l1off maddr l2e) ↦a {q4} l1e ∗ ⌜ addr_L1(va,l1e) = paddr ⌝.
```

Listing 4.1 A Strong Virtual Points-to Relation

4.2 A Restrictive Virtual Memory Addressing

A natural definition for a virtual points-to that depends on the current address space is to require that in order for a virtual address va to point to a value v, the assertion contains partial ownership of the physical memory involved in the page table walk that would translate va to its backing physical location — with locations existentially quantified since a virtual points-to should not assert *which* locations are accessed in a page table walk. Figure 4.1 gives such a definition. It asserts the existence of four page table entries, one at each translation level, and via L4_L1_PointsTo asserts from the current cr3 value, taking the offsets from va (as in Figure 3.1) to index each level, finding the existentially quantified entries in each level, ending with the L1 entry, whose physical page base is added to the page offset of va to obtain the physical address — and there (addr_L1(va,l1e)) in physical memory, is the value. Most of the definition lives directly in vProp, using the separation logic structure lifted from Iris's iProp.

L4_L1_PointsTo works by chaining together the entries for each level, using the sequence of table offsets from the address being translated to index each table level, and using the physical page address embedded in each entry.[2] For example, the first level address translation to get the L4 entry (l4e) uses the masks l4M52 with the current cr3 to get the physical address of the start of the L4 table, and l4off with the virtual address being translated to compute the correct offset within that table.

[2]The fractions q1 through q4 represent the fractional ownership of each entry based on how many word-aligned addresses might need to share the entry.

Then at that physical location is the appropriate entry in the L4 table l4e. Subsequent levels of the page table walk work similarly. The statement of these assertions is simplified by the use of our split physical points-to assertions, since each level of tables is page-sized.[3] This helper definition is also a more explicit vProp, explicitly binding a value for cr3 and using it to start the translation process.

This solution is in fact very close to that of Kolanski and Klein [97], who define a separation logic from scratch in Isabelle/HOL, where the semantics of all assertions are functions from pairs of heaps and page table root values to booleans.[4] Our solution removes some restrictions in this definition by further abstracting the handling of address translation.

4.3 Aliasing/Sharing Physical Pages

The virtual points-to definition shown in Figure 4.1 is too strong to specify some operations a virtual memory manager may need to do, such as move one level of the page table to a different physical location while preserving all virtual-to-physical mappings.[5] The use of $\mathsf{L_4\_L_1\_PointsTo}$ in Figure 4.1's virtual points-to definition' stores knowledge of the page table walk's details with ownership of the backing physical memory. Updating any of these mappings (e.g., moving the page tables in physical memory, as in coallescing for superpages or hugepages) would require explicitly collecting all virtual points-to facts that traverse affected entries. It is preferable to permit the page tables themselves to be updated independently of the virtual points-to assertions, so long as those updates preserve the same virtual-to-physical translations.

Intuitively, the definition in Figure 4.1 is too strong because the virtual points-to assertion there tracks too much information: when writing programs that access memory via virtual addresses, most

[3] We do not address superpages and hugepages in this thesis.
[4] This was a typical explicit construction at the time; their work significantly predates Iris.
[5] x86-64 hardware, like other architectures, includes a feature (which we do not formalize assertions for) to replace an L1 page table address in an L2 entry with a pointer to a *larger* 2MB page (called super-pages), or replace an L2 page table address in an L3 entry with a pointer to a 1GB page (called huge-pages).

$$\mathsf{va} \mapsto_{\mathsf{v}} \{\mathsf{q}\}\ \mathsf{val} : \mathsf{vProp}\ \Sigma \triangleq \exists \mathsf{pa}.\ \exists \delta.\ \underbrace{(\lambda cr3val.\ cr3val \hookrightarrow^{\delta s}\ \delta)}_{\text{Find addr. space invariant}} * \underbrace{\mathsf{va} \hookrightarrow^{\delta}_{\mathsf{q}} \mathsf{pa}}_{\text{Ghost translation}} * \underbrace{\mathsf{pa} \mapsto_{\mathsf{p}} \{\mathsf{q}\}\ \mathsf{val}}_{\text{Physical location}}$$

Figure 4.1 Virtual-Pointsto for Sharing Pages

code does not care *which physical memory locations are involved in address translation*: it only cares that virtual address translation would succeed. The necessary information about the physical page table walk must still be tracked, but can be tracked separately from the virtual points-to assertion itself. In practice the decisions about which virtual addresses are valid rest not with code posessing a virtual address, but with the virtual memory manager — and its invariants.

We separate the physical page-table-walk from the virtual pointsto relation, replacing it with ghost state that merely guarantees the address translation would succeed. Iris includes a *ghost map* construction which we use to track mappings from virtual addresses to the physical addresses they translate to. The implementation details in terms of named invariants and various resource algebras are not important to our presentation, so we treat the maps abstractly. The map includes for each key in the map (i.e., each virtual address) a token $k \hookrightarrow^{\gamma} v$ whose ownership is required to update that key-value pair in the ghost map named γ. The existence of such a token implies that the actual map θ tracked by a corresponding $\mathsf{GhostMap}(\gamma, \theta)$ resource indeed maps k to v. These properties are captured by the Iris rules:[6]

$$\mathsf{GhostMap}(\gamma, \theta) * \mathsf{pa} \hookrightarrow^{\gamma} \mathsf{va} \Rrightarrow \mathsf{GhostMap}(\gamma, \theta[\mathsf{pa} \mapsto \mathsf{va}']) * \mathsf{pa} \hookrightarrow^{\gamma} \mathsf{va}'$$

$$\mathsf{GhostMap}(\gamma, \theta) * \mathsf{pa} \hookrightarrow^{\gamma} \mathsf{va} \twoheadrightarrow \ulcorner \theta(\mathsf{pa}) = \mathsf{Some}(\mathsf{va}) \urcorner$$

Then the *virtual memory manager's invariant* ensures that for each $va \hookrightarrow^{\gamma} pa$ mapping in this map, there are *physical* resources sufficient to ensure that address translation for va will resolve in the hardware to pa via — $\mathsf{L_4\_L_1\_PointsTo}$. This allows modification of the page tables themselves as long as the changes do not invalidate overall virtual-to-physical translation. Consistency follows from replacing the page table walk with ownership of the ghost map token for the relevant ghost map entry, preventing the overall translation from being changed separately from the virtual points-to.

[6] Iris ghost maps lack established notation in the literature, but the syntax we use captures the details of `iris.base_logic.lib.ghost_map`.

$$\mathcal{I}\mathsf{ASpace}(\theta, m) \triangleq \mathsf{ASpace\_Lookup}(\theta, m) * \ast_{(\mathsf{va},\mathsf{paddr})\in\theta} \exists\ (\mathsf{l4e}\ \mathsf{l3e}\ \mathsf{l2e},\ \mathsf{l1e},\ \mathsf{paddr}).\ \mathsf{L_4\_L_1\_PointsTo}(\mathsf{va},\ \mathsf{l4e},\ \mathsf{l3e},\ \mathsf{l2e},\ \mathsf{l1e},\ \mathsf{paddr})$$

$$\text{where } \mathsf{ASpace\_Lookup}(\theta, m) \triangleq \lambda\ \mathsf{cr3val}.\ \exists\gamma\ .\ulcorner m\ !!\ \mathsf{cr3val} = \mathsf{Some}\ \gamma\urcorner * \mathcal{A}\mathsf{bsPTableWalk}(\delta, \theta)$$

Figure 4.2 Global Address-Space Invariant with a fixed global map of address-space names m

For clarity we refer to the ghost map summarizing virtual-to-physical translations by

$$\mathcal{A}\mathsf{bsPTableWalk}(\delta, \theta) \triangleq \mathsf{GhostMap}(\delta, \theta)$$

(omitting δ for brevity when only one is in scope) and keep this in a per-address-space invariant described shortly. We then replace the physical $\mathsf{L_4\_L_1\_PointsTo}$ in the virtual points-to definition with ownership of the token $\mathsf{va} \hookrightarrow^{\delta} \mathsf{paddr}$, guaranteeing that ghost map contains an mapping from the virtual address (va) to a physical address (pa), and thus that the per-address-space invariant contains the physical resources that guarantee the hardware resolves the translation.

We place the authorative ownership of the ghost translation $\mathcal{A}\mathsf{PTableWalk}$ in a per-address-space invariant $\mathcal{I}\mathsf{ASpace}$ (Figure 4.2). $\mathcal{I}\mathsf{ASpace}$ alone allows changes to the page tables that preserve overall virtual-to-physical translations. When combined with the fragment stored in the virtual points-to (Figure 4.1), changes in the virtual-to-physical translations become possible.

Because our logic is the first to address the existence of multiple address spaces, the ghost map of virtual-to-physical translations (identified by the ghost name δ in Figure 4.1 *for each address space*) must be locateable from a representation of the specific address space: the current root page table pointer stored in cr3.

Figure 4.2 describes the resources for each address space. θ is the logical map from virtual addresses to the physical addresses they should translate to, for the *current* address space, corresponding to the currently installed page tables (the ones indicated by cr3). In Figure 4.1, the existentially-quantified δ is the ghost name for this map (again, for the current address space). m is the (full) logical map from the various page table roots to ghost names for the per-address-space mappings like θ. It is tracked

by a ghost map named by the ghost name δs in Figure 4.1; there is only one such map in the system.

$\mathcal{I}\mathsf{ASpace}(\Theta, m)$ is then the resources for the current address space, ensuring that the current address space is valid (has an entry in m whose corresponding authoritative map matches Θ), and asserting ownership of the appropriate fractional ownership of the physical page table walk for each virtual-to-physical translation in Θ (for the current address space).

4.3.1 From A Single Address Space to Many

In principle the resources from each individual address space should be collected into a single shared invariant, and for each memory access, a fragment of this corresponding to $\mathcal{I}\mathsf{ASpace}(\Theta, m)$ for the current address space should be extracted from this global resource, used to prove correctness of the memory access, and put back. In this thesis we focus on the middle section, explicity identifying resources for each address space. The reason for this is that in practice, this global resource is also the place where kernel-specific assumptions, such as guaranteeing that certain virtual address range was mapped in all address spaces, would be enforced. This thesis focuses on the reasoning principles behind the virtual memory access, and we leave the use of this within a larger kernel to future work. We do, however, deal with the existence of multiple address spaces in examples (Section 5).

4.4 Address-Space Management

So far, we have introduced logical abstractions for a single address space, but VMMs handle more than one address-space, and doing so requires a way to talk about other address spaces, and means to switch address spaces.

$$[r](\mathsf{P}) : \mathsf{vProp}\ \Sigma \triangleq \lambda\_,\ \mathsf{P}\ \mathsf{r} \qquad \mathsf{Fact}\ \mathsf{P} \triangleq \forall\ , r\ r'.\ \mathsf{P}\ \mathsf{r} \dashv\vdash \mathsf{P}\ \mathsf{r'} \qquad \overline{\mathsf{Fact}\ [r](\mathsf{P})} \qquad \overline{\mathsf{Fact}\ (\mathsf{r} \mapsto_{\mathsf{r}} \{q\}\ \mathsf{rv})}$$

$$\overline{\mathsf{Fact}\ (\mathsf{w} \mapsto_{\mathsf{p}} \{q\}\ \mathsf{v})} \qquad \overline{(\mathsf{P} \vdash \mathsf{Q}) \vdash ([r](\mathsf{P}) \vdash [r](\mathsf{Q}))} \qquad \overline{[r](\mathsf{P} * \mathsf{Q}) \dashv\vdash ([r](\mathsf{P}) * [r](\mathsf{Q})}$$

$$\overline{[r](\mathsf{P} \wedge \mathsf{Q}) \dashv\vdash [r](\mathsf{P}) \wedge [r](\mathsf{Q})} \qquad \overline{[r](\mathsf{P} \vee \mathsf{Q}) \dashv\vdash [r](\mathsf{P}) \vee [r](\mathsf{Q})} \qquad \overline{\mathsf{Fact}\ \mathsf{P} \vdash [r](\mathsf{P}) \dashv\vdash (\mathsf{P})}$$

Figure 4.3 Other-space Modality and Its Laws

Figure 4.3 gives the definition of our modal operator for asserting the truth of an modal (address-space-contingent) assertion *in another address space*, which we call the *other-space* modality. The definition itself is not particularly surprising — as our modal assertions are semantic predicates on a page table root (physical) address, the assertion $[r](P)$ is a modal assertion that ignores the (implicit) current page table root and evaluates the truth P as if r were the page table root. The novelty here is not in the details of the definition but in recognizing that this is the right way to deal with multiple address spaces, and working out how to support the interaction of multiple address spaces (discussed in the next section).

We can prove that this modality follows certain basic laws: that its truth is independent of the address space in which it is considered, it distributes over various logical connectives, and it follows the rule of consequence. We call vProp assertions whose truth is independent of the current address space Facts; these include other-space assertions, physical memory points-tos, and register assertions. Facts can move in and out of other-space modalities freely.

4.4.1 Subtleties of Changing Address Spaces Using Modalities

Operating systems on x86-64 change address spaces by updating cr3. Conceptually, we expect a proof rule for such an update to take move resources in and out of modalities for the old and new address spaces. This suggests a rule like the following, which captures the intuition of the address space change, but is subtly unsound under common assumptions.

Broken

$$\{P * \mathsf{cr3} \mapsto_r \ r_1 * r \mapsto_r \ r_2 * [r_2](Q)\}\texttt{mov \%cr3},\ r\{[r_1](P) * \mathsf{cr3} \mapsto_r r_2 * r \mapsto_r r_2 * Q\}$$

This rule takes all assertions not under a modality (P), and moves them under a modality for the old address space (which is where they were valid), while pulling assertions for the new, target address space out of a corresponding modality. (For clarity, we also omit the address space invariants, which must also be present.) Unfortunately, as stated, this interacts quite poorly with the traditional frame

rule. If we abbreviate the pre- and post-conditions above as Pre and Post:

$$\dfrac{\dfrac{}{\{\mathsf{Pre}\}\texttt{mov \%cr3},\ r\{\mathsf{Post}\}}\ \textsc{Broken}}{\{a \mapsto_{\mathsf{v}} x * \mathsf{Pre}\}\texttt{mov \%cr3},\ r\{a \mapsto_{\mathsf{v}} x * \mathsf{Post}\}}\ \textsc{Frame}$$

Notice that both the precondition and postcondition assert that $a \mapsto_{\mathsf{v}} x$ in the current address space, but we have no basis for concluding that address translation is preserved by the change of address space. So this derivation clearly leads to an unsound conclusion. The essential problem is that the frame rule is motivated by local reasoning about local updates, but a switch of address space is a *global* change that may invalidate information about virtual addresses. Thus framing around cr3 updates is unsound, which has important impacts on our formal development.

Switching to Hoare doubles resolves this problem because an underappreciated subtlety of Hoare doubles is that typically *there is no primitive frame rule.* Instead, each verification essentially includes a local frame that it passes to the next instructions (think continuation-passing style), giving each overall rule a *global* (rather than local) precondition. For most rules, this is not that important, but it does permit rules that have global effects on their preconditions.

This leads to our actual rule for cr3 updates, which is *almost*

$$\textsc{ChangeAddressSpace}\ \dfrac{\{[r_1](P) * cr3 \mapsto_r r_2 * r \mapsto_r r_2 * Q\}\ \overline{is}}{\{P * cr3 \mapsto_r r_1 * r \mapsto_r r_2 * [r_2](Q)\}\ \texttt{mov \%cr3},\ r;\ \overline{is}}$$

(Again, the actual rule in Section 4.5 includes the address space invariants.) Because the precondition on this rule is global, we avoid issues with framing. A more typical frame rule can be recovered from Hoare doubles in the typical way[31] with the added restriction that cr3 is restored to its pre-framing value before un-framing (in this case, the framed-out assertion is effectively stashed under an other-space assertion for the initial address space, and pulled out as part of restoring).

4.5 Selected Logical Rules

Per the discussion in Section 4.4.1, we define our logic using Hoare doubles:[7]

$$\{\Phi\}_{\mathsf{rtv}}\ \mathsf{e} : \mathsf{iProp}\ \Sigma := ((\mathsf{cr3} \mapsto_{\mathsf{r}} \mathsf{rtv} * \Phi)\ \mathsf{rtv}) \twoheadrightarrow \mathsf{WP}\ \mathsf{e}\ \{\_, \mathsf{True}\}$$

Our Hoare doubles $\{\Phi\}_{\mathsf{rtv}}\ \mathsf{e}$ state that the expression (i.e., sequence of instructions) e are safe to execute (will not fault) when executed with vProp precondition $\Phi * \mathsf{cr3} \mapsto_{\mathsf{r}} \mathsf{rtv}$. WP is Iris's own weakest precondition modality, unmodified[90]. Making rtv an explicit parameter to the double rather than simply using register assertions directly solves a technical problem by ensuring that the page table root used to evaluate the vProp (i.e., evaluating the assertion in the *current*) address space is feasible. [8]

The rest of this section describes the specifications of some selected AMD64 instructions in our logic. These rules and others (e.g., including accessing memory at an instruction-specified offset from a register value, which is common in most ISAs) can be found in our artifact. Each rule in Figure 4.4, is annotated with a root (i.e., cr3) address value (rtv), under which the resources mentioned in the specification are valid. In general, we use metavariables r_s and r_d to specify source and destination registers for each instruction and prefix various register value variables with rv. We sometimes use r_a to emphasize when a register is expected to hold an address used for memory access, though the figure also uses typical assembler conventions of specifying memory access operands by bracketing the register holding the memory address. Standard for Hoare doubles, there is a frame resource P in each rule for passing resources not used by the first instruction in sequence through to subsequent instructions. Our rules include tracking each instruction's memory address to track rip updates, which is critical for control transfer instructions (Figures 4.6 and 4.6). Our development also includes handling of the rflags register updates from arithmetic instructions in Figure 4.7. Most rules are otherwise standard

[7] This omits some low-level Iris details (stuckness, observations) that play no meaningful role in our development.

[8] Consider the difficulty of selecting the correct page table root value from an arbitrary opaque Φ, which may even existentially quantify the page table root. An alternative is to require Φ to have a syntactic form where we can directly extract the value of cr3, but this makes using Iris Proof Mode (IPM) [101] with vProps difficult; IPM works for any type matching the signature of an Iris bi, which includes vProps, but manually guiding IPM to put an assertion in a specific position over and over adds significant proof burden.

$$\textsc{WriteToRegFromVirtMem}\quad \frac{\left\{\begin{array}{l} P * \mathcal{I}\mathsf{ASpace} * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_{\mathsf{r}} \mathsf{iv} + \mathsf{MovLen}(\mathsf{r}_d, [\mathsf{r}_a]) * \\ r_d \mapsto_r \mathsf{v} * r_a \mapsto_r \{q\}\ \mathsf{vaddr} * \mathsf{vaddr} \mapsto_{\mathsf{v,rtv}} \mathsf{v} \end{array}\right\}_{\mathsf{rtv}}\ \overline{is}}{\{P * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_{\mathsf{r}} \mathsf{iv} * r_d \mapsto_r \mathsf{rvd} * r_a \mapsto_r \{q\}\ \mathsf{vaddr} * \mathsf{vaddr} \mapsto_{\mathsf{v}} \mathsf{v}\}_{\mathsf{rtv}}\ \mathsf{mov}\ \mathsf{r}_d,\ [\mathsf{r}_a];\ \overline{is}}$$

$$\textsc{WriteToVirtMemFromReg}\quad \frac{\{P * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_{\mathsf{r}} \mathsf{iv} + \mathsf{MovLen}(\mathsf{r}_d, [\mathsf{r}_a]) * r_s \mapsto_r \{q\}\mathsf{rvs} * r_a \mapsto_r \{q\}\ \mathsf{vaddr} * \mathsf{vaddr} \mapsto_{\mathsf{v}} \mathsf{v}\}_{\mathsf{rtv}}\ \overline{is}}{\{P * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_{\mathsf{r}} \mathsf{iv} * r_s \mapsto_r \{q\}\mathsf{rvs} * r_a \mapsto_r \{q\}\ \mathsf{vaddr} * \mathsf{vaddr} \mapsto_{\mathsf{v}} \mathsf{rvs}\}_{\mathsf{rtv}}\ \mathsf{mov}\ [\mathsf{r}_a],\ \mathsf{r}_s;\ \overline{is}}$$

$$\textsc{WriteToRegCtlFromRegModal}\quad \frac{\{[\mathsf{rtv}](P * \mathcal{I}\mathsf{ASpace}) * \mathsf{rip} \mapsto_{\mathsf{r}} \mathsf{iv} + 4 * \mathcal{I}\mathsf{ASpace} * R * r_s \mapsto_r \{q\}\ \mathsf{rvs}\}_{\mathsf{rvs}}\ \overline{is}}{\{P * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_{\mathsf{r}} \mathsf{iv} * [\mathsf{rvs}](R * \mathcal{I}\mathsf{ASpace}) * r_s \mapsto_r \{q\}\ \mathsf{rvs}\}_{\mathsf{rtv}}\ \mathsf{mov}\ \mathsf{cr3},\ r_s;\ \overline{is}}$$

Figure 4.4 Reasoning Rules for Selected AMD64 Instructions under

(e.g., mov between registers, etc.), with Figure 4.4 showing the rules most unique to our development.

4.5.1 Accessing Virtual Addresses

Figure 4.4 includes two rules for accessing memory at an address stored in a register r_a. Ultimately, any memory access needs to ensure the relevant address translation would succeed, which can be ensured by what we informally call the page-table-walk points-to collection ($\mathsf{L_4\_L_1\_PointsTo}$ in Figure 4.1). WriteToRegFromVirtMem and WriteToVirtMemFromReg each use a virtual-pointsto assertion ($\mathsf{vaddr} \mapsto_{\mathsf{v}} \mathsf{v}$), and are nearly-standard (assembly) separation logic rules for memory accesses. However, because we split the physical resources for the page table walk from the virtual points-to itself (in order to permit the physical page tables to be freely modified as long as they preserve virtual-to-physical translations), the rule requires $\mathcal{I}\mathsf{ASpace}$ for the current address space to be carried through. The soundness proofs for these rules extract the token ($\mathsf{va} \hookrightarrow^{\delta}_{\mathsf{q}} \mathsf{pa}$) from the virtual points-to, use that to extract the physical page-table-traversal points-to collection describing the page table walk for the relevant address ($\mathsf{L_4\_L_1\_PointsTo}$) from the invariant ($\mathcal{I}\mathsf{ASpace}$), prove that the page table walk succeeds and memory or registers are updated appropriately, before re-packing the invariant and virtual points-to resources.

$$\textsc{PushReg64}\quad \frac{\left\{\ Q * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_{\mathsf{r}} \mathsf{iv} + \mathsf{PushLen} * \mathsf{rsp} \mapsto_{r} \mathsf{maddr} * \mathsf{maddr\text{-}8} \mapsto_{\mathsf{v}} \mathsf{rdv} * \mathsf{r} \mapsto_{\mathsf{r}} \mathsf{rdv}\ \right\}_{\mathsf{rtv}}\ \overline{is}}{\left\{\ P * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_{\mathsf{r}} \mathsf{iv} * \mathsf{rsp} \mapsto_{r} \mathsf{maddr\text{-}8} * \mathsf{maddr\text{-}8} \mapsto_{\mathsf{v}} \mathsf{mv} * \mathsf{r} \mapsto_{\mathsf{r}} \mathsf{rdv}\ \right\}_{\mathsf{rtv}}\ \mathsf{push\ r};\ \overline{is}}$$

$$\textsc{PopReg64}\quad \frac{\left\{\ Q * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_{\mathsf{r}} \mathsf{iv} + \mathsf{PopLen} * \mathsf{rsp} \mapsto_{r} \mathsf{maddr{+}8} * \mathsf{maddr} \mapsto_{\mathsf{v}} \mathsf{mv} * \mathsf{r} \mapsto_{\mathsf{r}} \mathsf{mv}\ \right\}_{\mathsf{rtv}}\ \overline{is}}{\left\{\ P * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_{\mathsf{r}} \mathsf{iv} * \mathsf{rsp} \mapsto_{r} \mathsf{maddr} * \mathsf{maddr} \mapsto_{\mathsf{v}} \mathsf{mv} * \mathsf{r} \mapsto_{\mathsf{r}} \mathsf{rdv}\ \right\}_{\mathsf{rtv}}\ \mathsf{pop\ r};\ \overline{is}}$$

Figure 4.5 Selected Reasoning Rules for Stack Pop and Push Instructions

4.5.2 Updating cr3

Unlike other rules, WriteToRegCtlFromRegModal updates the root address of the address-space determining the validity of resources, from rtv before the mov to rvs afterwards. The global effects of this rule are reflected in moving bare assertions under an other-space modality for rtv, and moving the new address space's assertions out of the corresponding modality.

4.5.3 Stack Operations: Push and Pop

In Figure 4.5, we give proof rules for the stack operations. The rest of the proof rules presented, including the stack-related ones, are simply lifting of the register and memory mappings used for reduction rules (in Section 3.4) into the ownership assertions (points-to assertion). For the operations pushing and popping the stack, the rules presented enforce proper value changes in the register of the stack pointer (rsp) and at the memory address kept in the register of the stack pointer (maddr and maddr-8).

4.5.4 Control-Flow Operations: Call Return and Jump

Regarding the specification of the control flow instructions presented in Figure 4.6, the essential piece is the predicate lifting of assurance of instructions mapped to the memory (CodeAt and CodeAtCond in Section 3.5.6).

Predicate PHoldsAtTarget asserts not only the fact that the loaded instruction sequence $\overline{iscall}$ can be

$$\textsc{Call}\ \frac{\left\{\ Q' * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_r \mathsf{iv} + \mathsf{CallLen} * \mathsf{rsp} \mapsto_r \mathsf{maddr} * \mathsf{r} \mapsto_r \mathsf{off}\ \right\}_{\mathsf{rtv}}\ \overline{is}}{\left\{\begin{array}{l} P * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{rsp} \mapsto_r \mathsf{maddr} * \mathsf{maddr\text{-}8} \mapsto_v \mathsf{mv} * \mathsf{r} \mapsto_r \mathsf{off} \\ \mathsf{PHoldsAtTarget}\ \gamma\ \overline{iscall}\ \overline{is}\ Q\ \mathsf{off}\ \mathsf{iv}\ \mathsf{maddr} \end{array}\right\}_{\mathsf{rtv}}\ \mathsf{call\ r};\ \overline{is}}$$

$$\textsc{Return}\ \frac{\left\{\ Q * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_r \mathsf{mv} * \mathsf{rsp} \mapsto_r \mathsf{maddr\text{-}8} * \mathsf{maddr} \mapsto_v \mathsf{mv}\ \right\}_{\mathsf{rvs}}\ \overline{is}}{\left\{\begin{array}{l} Pn * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{rsp} \mapsto_r \mathsf{maddr} * \mathsf{maddr} \mapsto_v \mathsf{mv} \\ \mathsf{PCodeAt}\ \gamma\ \overline{is}\ \mathsf{mv}\ \mathsf{maddr} \end{array}\right\}_{\mathsf{rtv}}\ \mathsf{ret}\ \overline{is}}$$

$$\textsc{JumpE}\ \frac{\left\{\ Pn * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_r \mathsf{iv} * \ulcorner \mathsf{flags\_set\_contains\ flaglist\ ZF} = \mathsf{false} \urcorner * \mathsf{fl\ rflags} \mapsto_r \mathsf{flg\ flagl'} * \mathsf{ZF} \mapsto_f 0\ \right\}_{\mathsf{rtv}}\ \overline{is}}{\left\{\ P * \mathcal{I}\mathsf{ASpace} * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{fl\ rflags} \mapsto_r \mathsf{flg\ flagl} * \mathsf{ZF} \mapsto_f \mathsf{zv} * \mathsf{PCodeAtCond}\ \gamma\ \mathsf{Pn} * \mathsf{off\ iv\ flagl}\ \right\}_{\mathsf{rtv}}\ \mathsf{jmp\ off};\ \overline{is}}$$

Figure 4.6 Selected Reasoning Rules for Call, Return and Jump Instructions

loaded to be executed with the resources in Q

$$Q\ \mathrel{-\!\!*}\ \mathsf{WP}\ \overline{iscall}$$

but also the assurance of the instructions after the return from the call: a part of the Q has to be PCodeAt in Rule Return rule. These predicates, as expected in the operational semantics as well, also connect the instruction sequences to the byte-stream encodings through *virtual byte-points-to* assertions: $ip \mapsto_b$ es asserting the fact that the instruction address ip points-to the first byte of the bytes stream es when ip is the value of current instruction pointer rip when the instruction sequence $\overline{is}$ is loaded.

For handling the flags as the assertion level, we introduce *flags-points-to* assertions from the flag identifiers to the value indicating whether they are set or not

$$\mathsf{ZF} \mapsto_f 0$$

4.5.5 Arithmetic & Bitwise Arithmetic Operations

Other than lifting the facts in operational semantics rules in Sections 3.5.4 and 3.5.3 to their counterparts as logical points-to assertion, proof rules for arithmetic and bitwise instructions in

Figure 4.7 do not exhibit any additional important aspect than what we discuss about them in Sections 3.5.4 and 3.5.3.

$$\textsc{ShiftLeftImm8}\quad \frac{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} + \mathsf{ShlLenRImm} * \mathsf{flg}\ \mathsf{rflags} \mapsto_r \mathsf{flagl'} * \mathsf{r} \mapsto_r \{q\}\ \mathsf{rv} << \mathsf{imm}\ \}_{\mathsf{rtv}}\ \overline{is}}{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{flg}\ \mathsf{rflags} \mapsto_r \mathsf{flagl} * \mathsf{r} \mapsto_r \{q\}\ \mathsf{rv}\}_{\mathsf{rtv}}\ \mathsf{shl}\ \mathsf{r}, \mathsf{imm};\ \overline{is}}$$

$$\textsc{ShiftRightImm8}\quad \frac{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} + \mathsf{ShlLenRImm} * \mathsf{flg}\ \mathsf{rflags} \mapsto_r \mathsf{flagl'} * \mathsf{r} \mapsto_r \{q\}\ \mathsf{rv} >> \mathsf{imm}\ \}_{\mathsf{rtv}}\ \overline{is}}{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{flg}\ \mathsf{rflags} \mapsto_r \mathsf{flagl} * \mathsf{r} \mapsto_r \{q\}\ \mathsf{rv}\}_{\mathsf{rtv}}\ \mathsf{shr}\ \mathsf{r}, \mathsf{imm};\ \overline{is}}$$

$$\textsc{OrLeftImm32}\quad \frac{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} + \mathsf{ShlLenRImm} * \mathsf{flg}\ \mathsf{rflags} \mapsto_r \mathsf{flagl'} * \mathsf{r} \mapsto_r \{q\}\ \mathsf{rv}||\mathsf{imm}\ \}_{\mathsf{rtv}}\ \overline{is}}{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{flg}\ \mathsf{rflags} \mapsto_r \mathsf{flagl} * \mathsf{r} \mapsto_r \{q\}\ \mathsf{rv}\}_{\mathsf{rtv}}\ \mathsf{or}\ \mathsf{r}, \mathsf{imm};\ \overline{is}}$$

$$\textsc{AndLeftImm32}\quad \frac{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} + \mathsf{ShlLenRImm} * \mathsf{flg}\ \mathsf{rflags} \mapsto_r \mathsf{flagl'} * \mathsf{r} \mapsto_r \{q\}\ \mathsf{rv}\&\mathsf{imm}\ \}_{\mathsf{rtv}}\ \overline{is}}{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{flg}\ \mathsf{rflags} \mapsto_r \mathsf{flagl} * \mathsf{r} \mapsto_r \{q\}\ \mathsf{rv}\}_{\mathsf{rtv}}\ \mathsf{and}\ \mathsf{r}, \mathsf{imm};\ \overline{is}}$$

$$\textsc{AndLeftReg64}\quad \frac{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} + \mathsf{AndLenRR} * \mathsf{flg}\ \mathsf{rflags} \mapsto_r \mathsf{flagl'} * \mathsf{r}_{\mathsf{src}} \mapsto_r \{q\}\ \mathsf{src} * \mathsf{r}_{\mathsf{dst}} \mapsto_r\ \mathsf{dst}\&\ \mathsf{src}\}_{\mathsf{rtv}}\ \overline{is}}{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{flg}\ \mathsf{rflags} \mapsto_r \mathsf{flagl} * \mathsf{r}_{\mathsf{src}} \mapsto_r \{q\}\ \mathsf{src} * \mathsf{r}_{\mathsf{dst}} \mapsto_r\ \mathsf{dst}\}_{\mathsf{rtv}}\ \mathsf{and}\ \mathsf{r}_{\mathsf{src}}, \mathsf{r}_{\mathsf{dst}};\ \overline{is}}$$

$$\textsc{OrLeftReg64}\quad \frac{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} + \mathsf{OrLenRR} * \mathsf{flg}\ \mathsf{rflags} \mapsto_r \mathsf{flagl'} * \mathsf{r}_{\mathsf{src}} \mapsto_r \{q\}\ \mathsf{src} * \mathsf{r}_{\mathsf{dst}} \mapsto_r\ \mathsf{dst}||\ \mathsf{src}\}_{\mathsf{rtv}}\ \overline{is}}{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{flg}\ \mathsf{rflags} \mapsto_r \mathsf{flagl} * \mathsf{r}_{\mathsf{src}} \mapsto_r \{q\}\ \mathsf{src} * \mathsf{r}_{\mathsf{dst}} \mapsto_r\ \mathsf{dst}\}_{\mathsf{rtv}}\ \mathsf{or}\ \mathsf{r}_{\mathsf{src}}, \mathsf{r}_{\mathsf{dst}};\ \overline{is}}$$

$$\textsc{AddReg64Imm32}\quad \frac{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{r}_d \mapsto_r \mathsf{w} * \mathsf{overflow}(\mathsf{dst} + \mathsf{imm}) = (w, b) * \mathsf{fl}\ \mathsf{rflags} \mapsto_r \mathsf{flg}\ \mathsf{flagl}\}_{\mathsf{rtv}}\ \overline{is}}{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{r}_{\mathsf{dst}} \mapsto_r \mathsf{dst} * \mathsf{fl}\ \mathsf{rflags} \mapsto_r \mathsf{flg}\ \mathsf{flagl}\}_{\mathsf{rtv}}\ \mathsf{add}\ \mathsf{r}_d, \mathsf{imm};\ \overline{is}}$$

$$\textsc{AddReg64Reg64}\quad \frac{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{overflow}(\mathsf{dst} + \mathsf{src}) = (w, b) * \mathsf{r}_{\mathsf{dst}} \mapsto_r \mathsf{w} * \mathsf{r}_{\mathsf{src}} \mapsto_r \{q\}\mathsf{src} * \mathsf{fl}\ \mathsf{rflags} \mapsto_r \mathsf{flg}\ \mathsf{flagl}\}_{\mathsf{rtv}}\ \overline{is}}{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{r}_{\mathsf{dst}} \mapsto_r \mathsf{rvd} * \mathsf{r}_{\mathsf{src}} \mapsto_r \{q\}\mathsf{src} * \mathsf{fl}\ \mathsf{rflags} \mapsto_r \mathsf{flg}\ \mathsf{flagl}\}_{\mathsf{rtv}}\ \mathsf{add}\ \mathsf{r}_{\mathsf{dst}}, \mathsf{r}_{\mathsf{src}};\ \overline{is}}$$

$$\textsc{SubReg64Imm32}\quad \frac{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{r}_d \mapsto_r \mathsf{w} * \mathsf{overflow}(\mathsf{dst}\ \text{-}\ \mathsf{imm}) = (w, b) * \mathsf{fl}\ \mathsf{rflags} \mapsto_r \mathsf{flg}\ \mathsf{flagl}\}_{\mathsf{rtv}}\ \overline{is}}{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{r}_{\mathsf{dst}} \mapsto_r \mathsf{dst} * \mathsf{fl}\ \mathsf{rflags} \mapsto_r \mathsf{flg}\ \mathsf{flagl}\}_{\mathsf{rtv}}\ \mathsf{sub}\ \mathsf{r}_d, \mathsf{imm};\ \overline{is}}$$

$$\textsc{SubReg64Reg64}\quad \frac{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{overflow}(\mathsf{dst}\ \text{-}\ \mathsf{src}) = (w, b) * \mathsf{r}_{\mathsf{dst}} \mapsto_r \mathsf{w} * \mathsf{r}_{\mathsf{src}} \mapsto_r \{q\}\mathsf{src} * \mathsf{fl}\ \mathsf{rflags} \mapsto_r \mathsf{flg}\ \mathsf{flagl}\}_{\mathsf{rtv}}\ \overline{is}}{\{P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{r}_{\mathsf{dst}} \mapsto_r \mathsf{rvd} * \mathsf{r}_{\mathsf{src}} \mapsto_r \{q\}\mathsf{src} * \mathsf{fl}\ \mathsf{rflags} \mapsto_r \mathsf{flg}\ \mathsf{flagl}\}_{\mathsf{rtv}}\ \mathsf{sub}\ \mathsf{r}_{\mathsf{dst}}, \mathsf{r}_{\mathsf{src}};\ \overline{is}}$$

$$\textsc{CmpReg64Reg64}\quad \frac{\left\{\begin{array}{l} P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{r}_{\mathsf{dst}} \mapsto_r \mathsf{dst} * \mathsf{r}_{\mathsf{src}} \mapsto_r \{q\}\mathsf{src} * \mathsf{fl}\ \mathsf{rflags} \mapsto_r \mathsf{flg}\ \mathsf{flagl'} * \\ \left\{\ (\ulcorner \exists \mathsf{p}, (\mathsf{dst}\ \text{-}\ \mathsf{src})) = \mathsf{N.pos}\ \mathsf{p}\urcorner * * \mathsf{ZF} \mapsto_{\mathsf{f}} 0) \vee (\ulcorner (\ \mathsf{dst}\ \text{-}\ \mathsf{src}) = 0\urcorner * \mathsf{ZF} \mapsto_{\mathsf{f}} 1)\ \right\} \end{array}\right\}_{\mathsf{rtv}}\ \overline{is}}{\left\{\ P * \mathsf{rip} \mapsto_r \mathsf{iv} * \mathsf{r}_{\mathsf{dst}} \mapsto_r \mathsf{rvd} * \mathsf{r}_{\mathsf{src}} \mapsto_r \{q\}\mathsf{src} * \mathsf{fl}\ \mathsf{rflags} \mapsto_r \mathsf{flg}\ \mathsf{flagl} * \mathsf{ZF} \mapsto_{\mathsf{f}} \mathsf{bzf})\ \right\}_{\mathsf{rtv}}\ \mathsf{cmp}\ \mathsf{r}_{\mathsf{dst}}, \mathsf{r}_{\mathsf{src}};\ \overline{is}}$$

Figure 4.7 Selected Reasoning Rules for Arithmetic & Bitwise Instructions

4.6 Soundness

Our rules from Figures in Section 4.5 are proven sound in Iris against an assembly-level hardware model implementing a fragment of x86-64 including 64-bit address translation with 4-level page tables. Our rules for control transfers (jne, call, and ret) are currently axiomatized (with completely standard specifications[31,130]) because Iris's built-in machinery does not provide convenient ways to discard the current continuation; adaptation of others' approaches[45] is ongoing work. Our soundness proofs for all other instructions (including, critically, all memory accesses) are axiom-free.

CHAPTER 5

VERIFYING VMM ESSENTIALS

To both validate and demonstrate the value of the modal approach to reasoning about virtual memory management, we study several distillations of real concerns of virtual memory managers. Recall from Section 4 that virtual points-to assertions work just like regular points-to assertions, by design. In this section we work through two critical and challenging aspects of virtual memory management. First, in several stages, we work up to mapping a new page in the current address space. This requires a number of challenging substeps: dynamically traversing a page table to find the appropriate L1 entry to update; inserting additional levels of the page table if necessary (updating the VMM invariants along the way); converting the physical addresses found in intermediate entries into the corresponding virtual addresses that can be used for access; installing the new mapping; and collecting sufficient resources to form a virtual points-to assertion. Of these, only the second-to-last step (installing the correct mapping into the current address space) has previously been formally verified with respect to a machine model with address translation. Second, we formally verify a switch into a new address space as part of a task switch.

5.1 Traversing Live Page Tables

We build up to the main task of mapping a new page after traversing page tables in software. The mapping operation of Figure 5.5 assumes an operation walkpgdir which must traverse the page tables in order to locate the address of the L1 entry to update—possibly allocating tables for levels 3, 2, and 1 in the process, installing them into levels 4, 3, and 2, along the way. Traversing the page tables

is itself challenging functionality to verify: loading the current table root from cr3 is straightforward (a mov instruction), however, this produces the physical address of cr3, not the virtual address the kernel code would use to access that memory. This problem repeats at each level of the page table: assuming the code has *somehow* read the appropriate L4 (or L3, or L2) entry, those entries again yield physical addresses, not virtual ones.

5.1.1 Loading Page-Table Address Value

We will discuss access to the level 4 table later (Section 5.1.5). But for subsequent levels, the base address of level n must be fetched from the appropriate entry in the level $n+1$ table. This is the role of pte_get_next_table (Figures 5.1 and 5.3): it is passed the virtual address of the page table entry in level $n+1$, and should return the *virtual* address of the *base* of the level n table indicated by that entry. If the entry is empty (i.e., this is a sparse part of the page table representation), the code also allocates a page for the level n table, installs it in the level $n+1$ entry, and establishes appropriate invariants. Figure 5.1 presents the initial part of the function, which performs the allocation if necessary. Figure 5.3 (discussed in Section 5.1.4) deals with the cases where no allocation is necessary *or* the allocation has already been performed by the code in this figure.

Note that none of the verification for this function assumes specific page table levels — logical parameter v represents the level of the entry passed as an argument, and this code is used for all three level transitions when traversing page tables (4 to 3, 3 to 2, 2 to 1). This comes into play with a subtlety of the specification of pte_get_next_table that we will revisit several times: pte_get_next_table 's specification assumes it is given a virtual *vpte-pointsto*[1] granting access to the specified entry, but its postcondition does not yield new virtual points-to assertions! Instead it merely computes the base address of the next table, and returns adequate capabilities (discussed in Section 5.1.2) for the *caller* to construct a vpte-pointsto for the next table level (if this is not an L1 entry — the caller knows which level of the table this is for).

[1] A *vpte-pointsto* va $\mapsto_{\text{vpte,qfrac}}$ pa v is a virtual points-to granting access to virtual address va, which is known to hold v, but additionally exposing the physical address pa that the virtual address translates to. This is commonly used, as the name suggests, when updating PTEs, where we require assurance that we are accessing the intended PTE. Its definition is just like the regular virtual points-to in Figure 4.1, except taking the physical address as a parameter rather than existentially quantifying it.

Within get_next_table, after a standard function prologue, the code loads the entry entry pointed to by the argument. This is a page table entry: a 64-bit word divided into bit-fields for the physical address of the next table, and control bits like the valid bit, as discussed in Section 3.1.

Lines 19–21 check if the entry's "present" bit is set. If it is zero, a new page must be allocated for the next level of the table — which is done by the fall-through from Line 22's conditional jump. Otherwise the code jumps ahead to the case for the next level already existing, which is discussed in Section 5.1.4 and Figure 5.3. First, we must discuss another refinement of the address space invariant, establishing enough structure on the page tables themselves to allow the traversal. The code for allocating a new level of the page table must establish this extended invariant.

5.1.2 Identity Mappings

It is typical for kernels to need to convert between physical and virtual addresses, in both directions. Traversing the page tables in software is the simplest way to convert a virtual address to a physical address; this is the context we are working up to. However, implementing this virtual-to-physical (V2P) translation in this way ironically requires physical-to-virtual (P2V) translation, because the addresses stored in page table entries are physical, but memory accesses issued by the OS code use virtual addresses. There is no universal way to convert physical addresses to virtual — doing so relies on the kernel maintaining careful invariants or additional data structures to enable P2V translation.

In practice, VMM operations are performance-critical for many workloads, so most kernels opt for using invariants to make P2V conversion very fast, rather than maintaining yet another data structure. Most kernels maintain an invariant on their page tables that the virtual address of any page used for a page table lives at a virtual address whose value is *a constant offset from the physical address* — a practice sometimes referred to as *identity mapping* (even though the physical-to-virtual translation is typically not literally the identity function, but adding a non-zero constant offset).[2]

For this reason we extend the per-address-space invariant as in Figure 5.1, to also track which

[2]Some kernels do this for all physical memory on the machine, simplifying interaction with DMA devices. On newer platforms like RISC-V, this sometimes truly is an identity mapping — x86-64 machines are forced into offsets by backwards compatibility with bootloaders that cannot access the full memory space of the machine.

```
1  ;;pte_t *pte_get_next_table(pte_t *entry) {
2  ... ;; setting up the stack
3  ;; pte_t *next;
4  {P * IASpace_id(θ, Ξ \ {entry}, m) * rbp-8 ↦_v entry * rbp-16 ↦_v next * r8 ↦_r _ * rdi ↦_r _ * rtv ↪^δs δ}_rtv
5  {entry+KERNBASE ↦_vpte,qfrac entry entry_val * ⌜qfrac = 1 ↔ ¬(entry_present entry_val)⌝}_rtv
6  {⌜entry_present(entry_val)⌝ -* ∀_i∈0..511 table_root(entry_val.pfn) + i * 8 ↪^id v-1}_rtv
7  mov −0x8[rbp],rdi
8  {P * IASpace_id(θ, Ξ \ {entry}, m) * rbp-8 ↦_v entry * r8 ↦_r _ * rbp-16 ↦_v next * rdi ↦_r entry * rtv ↪^δs δ}_rtv
9  {entry+KERNBASE ↦_vpte,qfrac entry entry_val * ⌜qfrac = 1 ↔ ¬(entry_present entry_val)⌝}_rtv
10 {⌜entry_present(entry_val)⌝ -* ∀_i∈0..511 table_root(entry_val.pfn) + i * 8 ↪^id v-1}_rtv
11 mov rdi, r8
12 {P * IASpace_id(θ, Ξ \ {entry}, m) * rbp-8 ↦_v entry * r8 ↦_r entry * rbp-16 ↦_v next * rdi ↦_r entry * rtv ↪^δs δ}_rtv
13 {entry+KERNBASE ↦_vpte,qfrac entry entry_val * ⌜qfrac = 1 ↔ ¬(entry_present entry_val)⌝}_rtv
14 {⌜entry_present(entry_val)⌝ -* ∀_i∈0..511 table_root(entry_val.pfn) + i * 8 ↪^id v-1}_rtv
15 mov [r8],rdi
16 {P * IASpace_id(θ, Ξ \ {entry}, m) * rbp-8 ↦_v entry * r8 ↦_r entry * rbp-16 ↦_v next * rdi ↦_r entry_val * rtv ↪^δs δ}_rtv
17 {entry+KERNBASE ↦_vpte,qfrac entry entry_val * ⌜qfrac = 1 ↔ ¬(entry_present entry_val)⌝}_rtv
18 {⌜entry_present(entry_val)⌝ -* ∀_i∈0..511 table_root(entry_val.pfn) + i * 8 ↪^id v-1}_rtv
19 and 0x1,rdi
20 mov rdi,rax
21 cmp 0x0,rax ;; if (!entry->present) {
22 jne 161 <pte_get_next_table+0xa1>     ;; Jump if the present bit is not zero
23 {P * IASpace_id(θ, Ξ \ {entry}, m) * rbp-8 ↦_v entry * r8 ↦_r entry * rbp-16 ↦_v next * rdi ↦_r entry_val & 0x1}_rtv
24 {entry ↦_id v * rtv ↪^δs δ * rax ↦_r entry_val & 0x1 * ⌜entry_val & 0x1 = 0x0⌝}_rtv
25 {entry+KERNBASE ↦_vpte,qfrac entry entry_val * ⌜qfrac = 1 ↔ ¬(entry_val & 0x1 = 0x1)⌝}_rtv
26 mov rbp,rdi
27 sub 0x10, rdi ;; Store the value of rbp minus 16 bytes (address of next) into rdi
28 {P * IASpace_id(θ, Ξ \ {entry}, m) * rbp-8 ↦_v entry * r8 ↦_r entry * rbp-16 ↦_v next * rdi ↦_r rbp - 16}_rtv
29 {entry ↦_id v * rtv ↪^δs δ}_rtv
30 {entry+KERNBASE ↦_vpte,qfrac entry entry_val * ⌜qfrac = 1 ∧ ¬(entry_present entry_val)⌝}_rtv
31 callq 70 <pte_initialize> ;;pte_initialize(entry);
32 {P * IASpace_id(θ, Ξ \ {entry}, m) * rbp-8 ↦_v entry * r8 ↦_r entry}_rtv
33 {entry ↦_id v * rtv ↪^δs δ}_rtv
34 {entry+KERNBASE ↦_vpte,qfrac entry entry_val * ⌜qfrac = 1 ∧ ¬(entry_present entry_val)⌝}_rtv
35 {⌜qfrac = 1 ∧ ¬(entry_present (pfn_set (entry_val nextpaddr))⌝}_rtv
36 {rbp-16 ↦_v pfn_set(entry_val nextpaddr)}_rtv
37 { entry_present (pte_initialized (pfn_set(entryv nextpaddr))) -*
     ∀_i∈0 ... 511. ((table_root (pte_initialized (pfn_set(entry_val nextpaddr)))) + i * 8) ↪^id v-1 }
38 ... ;;entry value updates: entry->pfn = nextpaddr; entry->present = 1;
39 ... ;;now we know that entry is initialized, so we satisfy the condition to access children list
40 {P * IASpace_id(θ, Ξ \ {entry}, m) * rbp-8 ↦_v entry * r8 ↦_r entry}_rtv
41 {entry ↦_id v * rtv ↪^δs δ}_rtv
42 {entry+KERNBASE ↦_vpte,qfrac pte_initialized(pfn_set(entryv nextpaddr)) * ⌜qfrac = 1 ∧ ¬(entry_present entry_val)⌝}_rtv

43 { rbp-16 ↦_v pte_initialized(pfn_set(entry_val nextpaddr))*
   rax ↦_r table_root (pte_initialized (pfn_set(entry_val nextpaddr))) }_rtv
44 {∀_i∈0 ... 511. ((table_root (pte_initialized (pfn_set(entry_val nextpaddr)))) + i * 8) ↪^id v-1}
45 ;;}
46 ... ;; Code after conditional continued in Figure 5.3
```

Listing 5.1 Ensuring entry points to a valid next table, allocating if necessary.

$$\mathcal{I}\mathsf{ASpace}_{\mathsf{id}}(\theta,\Xi,m) \triangleq \mathsf{ASpace\_Lookup}_{\mathsf{id}}(\theta,\Xi,m) * \mathsf{GhostMap}(\mathsf{id},\Xi) *$$
$$\left(\circledast_{(\mathsf{va},\mathsf{paddr})\in\theta} \exists\,(\mathsf{l4e},\,\mathsf{l3e},\,\mathsf{l2e},\,\mathsf{l1e},\,\mathsf{paddr}).\,\mathsf{L_4\_L_1\_PointsTo}(\mathsf{va},\,\mathsf{l4e},\,\mathsf{l3e},\,\mathsf{l2e},\,\mathsf{l1e},\,\mathsf{paddr})\right) *$$
$$\circledast_{(\mathsf{pa},\mathsf{level})\in\Xi} \exists\,(\mathsf{qfrac},\,\mathsf{q},\,\mathsf{val},\mathsf{va}).\,\ulcorner \mathsf{va} = \mathsf{pa} + \mathsf{KERNBASE}\ \mathsf{level} > 1\urcorner * \underbrace{\mathsf{va} \hookrightarrow^{\delta}_{\mathsf{q}} \mathsf{pa}}_{\text{Ghost translation}} * \underbrace{\mathsf{pa} \mapsto_{\mathsf{p}} \{\mathsf{qfrac}\}\ \mathsf{val}}_{\text{Physical location}} *$$
$$\underbrace{\ulcorner \mathsf{qfrac} = 1 \leftrightarrow\ \neg\mathsf{entry\_present}\ (\mathsf{val})\urcorner}_{\text{Entry validity}} * \underbrace{\left(\ulcorner \mathsf{present\_L}(\mathsf{val},\mathsf{level})\urcorner \mathrel{-\!\!*} \forall_{\mathsf{i}\in 0..511}.\,((\mathsf{entry\_page}\ \mathsf{val}) + \mathsf{i} * 8) \hookrightarrow^{\mathsf{id}} \mathsf{level\text{-}1}\right)}_{\text{Indexing into next level of tables}}$$

where

$$\mathsf{ASpace\_Lookup}_{\mathsf{id}}(\theta,\Xi,m) \triangleq \lambda\ \mathsf{cr3val}.\ \exists\gamma\,.\,\ulcorner m\ !!\ \mathsf{cr3val} = \mathsf{Some}\ \gamma\urcorner * \mathcal{A}\mathsf{bsPTableWalk}(\delta,\theta) * \mathcal{A}\mathsf{PVMappings}(\delta,\Xi)$$
$$\mathsf{present\_L}(\mathsf{val},\mathsf{level}) \triangleq \mathsf{entry\_present}(\mathsf{val}) \wedge \mathsf{level} > 0$$

Figure 5.1 Global Address-Space Invariant in Figure 4.2 extended with a ghost map bookkeeping identity mappings

addresses we can perform a P2V conversion on by a adding a constant offset. Ξ is another ghost map, from physical addresses to the level of the page table they represent (1–4). *Only* physical addresses in Ξ can undergo P2V conversion. We describe proof of such a conversion in Section 5.1.4, but describe the invariant here because installing a new level 3/2/1 table requires maintaining that invariant.

For each $\mathsf{pa} \mapsto \mathsf{v} \in \Xi$, the invariant tracks a virtual points-to justifying that virtual address $\mathsf{pa} + \mathsf{KERNBASE}$ maps to physical address pa (the "Ghost translation" in Figure 5.1); fractional ownership of the physical memory for that page table entry; and for valid entries (with the present bit set) above L1, ghost map tokens for every entry in the table pointed to by the entry, which can be used to repeat the process one level down. i (L1 entries point to data pages, whose physical memory ownership resides in some virtual points-to). The assertion on Line 6 originates from the invariant one level up, and if this code determines the valid bit is set, it can return those child tokens without the conditional guard.

The fractional ownership of the entry's physical memory is subtle. Recall that $\mathsf{L_4\_L_1\_PointsTo}$ retains some physical ownership of each page table entry that is traversed (proportional to how many virtual addresses share the entry). So in general the invariant cannot keep full permission to the memory in this part of the invariant, or it would overlap the page table walk for virtual points-to assertions. But in the case where the entry is invalid, we may need to write to it (e.g., to install a reference to a next-level table, as we do in Figure 5.1), which requires full permission.

Fortunately, the entry can only be in use if its valid bit is set; if the valid bit is not set we know that no virtual points-to assertions in δ/θ have any partial ownership. Thus we use the invariant portion annotated as "Entry validity" in Figure 5.1 to capture this: if the entry is invalid the invariant holds full ownership of the entry, so it can be updated; while if the entry is valid, the invariant owns only a constant non-zero fragment sufficient to read the entry, but not modify it (which would invalidate some virtual points-to assertions):

$$\ulcorner \mathsf{qfrac} = 1 \leftrightarrow \ \neg \mathsf{entry\_present}\ (\mathsf{val}) \urcorner \qquad (*)$$

Thus the fractional ownership of the physical location is enough for Line 15 in Figure 5.1 to access the entry, though in get_next_table the caller has pulled that piece of information out of the invariant and passed it for the entry at hand. This removal appears explicitly in assertions, as the argument to the invariant is $\Xi \setminus \{\mathsf{entry}\}$ (indexing by the set Ξ allows us to borrow the physical resources for a specific page table entry out of the invariant, and later put them back). Line 22's conditional then determines in the fall-through case that the bit is not set, which together with other facts entails $\mathsf{qfrac} = 1$ at Line 30, and permits storing a new entry (in ellided code around Line 38)

This seemingly-simple piece of code has a highly non-trivial correctness argument, which depends critically on detailed invariants on how access to page table entries is shared between parts of the kernel. No prior work has engaged with this problem.

5.1.3 Installing a New Table

After obtaining the identity mapping for entry, we are able to load the entry_val into rdi, and check the presence bit through Lines 19–21 in Figure 5.1. Accessing the presence bit and checking the value allows us to exploit the side condition (*) when verifying the allocation path for when the entry is invalid (Lines 26–44 in Figure 5.1). This operation is subtle: the operation requires that the relevant table entry is readable, but the exact portion of ownership returned must be determined by inspecting the valid bit of the value in memory — so full ownership is returned only for unused

```
1  {P ∗ IASpace_id(θ, Ξ, m) ∗ rbp-16 ↦_r _ ∗ rdi ↦_r entry+KERNBASE}_rtv
2  {rax ↦_r _ ∗ ¬(entry_present entry_val}_rtv
3  {entry+KERNBASE ↦_vpte,1 entry entryv ∗ (entry) ↪^id level}
4  ;;void pte_initialize(pte_t *entry) {
5  ... ;;set up the stack
6  ;;allocate a full zeroed page for 512 8−byte entries
7  callq 81 <kalloc> ;;pte_t *local = kalloc();
8  mov rax,−0x10[rbp] ;; Store into 'local'
9  ;;entry−>pfn = PTE_ADDR_TO_PFN((uintptr_t) local);
10 mov −0x10[rbp],rax
11 mov −0x8[rbp],rdi
12 mov rax,[rdi]
13 {P ∗ IASpace_id(θ, Ξ, m) ∗ rbp-16 ↦_r _ ∗ rdi ↦_r entry+KERNBASE}_rtv
14 {entry+KERNBASE ↦_vpte, entry entryv ∗ (entry) ↪^id level}
15 {rax ↦_r nextpaddr ∗ ¬(entry_present entry_val)}
16 {entry+KERNBASE ↦_vpte,1 entry pfn_set(entryv nextpaddr)}
17 {rtv ↪^δs δ ∗ ⌜entry_present (pte_initialize(pfn_set(entry_val nextpaddr)),level)⌝ −∗
                  ∀_i∈0..511 table_root (pte_initialized (pfn_set (entry_val nextpaddr))) + i * 8 ↦_id level-1 }_rtv
18 ... ;;clean up the stack, return
```

Listing 5.2 Allocating a physical page

entries. When the bit is not set, that entails full ownership of the entry's memory ($\mathsf{qfrac} = 1$) and justifies writing to that memory. Otherwise, the code jumps past the end of this listing, to the following code at the top of Figure 5.3 (which is also the continuation of this code).

If the entry is not set, pte_initialize is called (Line 31 in Figure 5.1) for a physical page (utilizing the page-allocator's kalloc – currently the only axiomatized call in the the proof of pte_initialize (Line 7 in Figure 5.2).

Since we are using pte_initialize for page-table address allocation, we must relate this newly allocated physical address to the identity mapping map Ξ — see Line 37 in Figure 5.1, where `kalloc`'s specification guarantees it has returned memory from a designated memory pool that is already mapped[3] and satisfies the offset invariants. The soundness argument of this specification relies on the fact that these freshly allocated resources are part of an entry construction that has not been completed yet: the presence bit is set (Line 38 in Figure 5.1) after these freshly allocated resources are incorporated to the entry construction via the page-frame portion of the PTE. In other words,

[3] A reasonable reader might wonder where this pool initially comes from, and how it might grow when needed. Typically an initial mapping subject to this identity mapping constraint is set up prior to transition to 64-bit kernel code (notably, a page table must exist *before* virtual memory is enabled during boot, as part of enabling it is setting a page table root). Growing this pool later requires cooperation of physical memory range allocation and virtual memory range allocation, typically by starting general virtual address allocation at the highest physical memory address plus the identity mapping offset. This reserves the virtual addresses corresponding to all physical addresses plus the offset for later use in this pool, as needed.

```
1  ... ;; Continued from Figure 5.1; assertions below specialized to non−allocating path for clarity
2  {P ∗ IASpace_id(θ, Ξ \ {entry}), m) ∗ rbp-8 ↦_v entry ∗ rcx ↦_r _}_rtv
3  {entry ↦_id _ ∗ rtv ↪^δs δ}_rtv
4  {entry+KERNBASE ↦_vpte,qfrac (pte_initialized (entry_val.pfn))^¬}_rtv
5  {rbp-16 ↦_v (pte_initialized (entry_val.pfn))) ∗ rax ↦_r table_root (pte_initialize(entry_val.pfn))}_rtv
6  {∀_i∈0 ... 511. ((table_root (pte_initialized (entry_val.pfn)))) + i * 8) ↪^id v-1}
7  ;;uintptr_t next_virt_addr = (uintptr_t) P2V(entry.pfn«12);
8   movabs KERNBASE,rcx
9   add rcx,rax
10 {P ∗ IASpace_id(θ, Ξ \ {entry}), m) ∗ rbp-8 ↦_v entry ∗ rcx ↦_r KERNBASE}_rtv
11 {entry ↦_id _ ∗ rtv ↪^δs δ}_rtv
12 {entry+KERNBASE ↦_vpte,qfrac (pte_initialized (entry_val.pfn))^¬}_rtv
13 {rbp-16 ↦_v (pte_initialized (entry_val.pfn))) ∗ rax ↦_r table_root (pte_initialize(entry_val.pfn)) + KERNBASE}_rtv
14 {∀_i∈0 ... 511. ((table_root (pte_initialized (entry_val.pfn)))) + i * 8) ↪^id v-1}
15 ...
16 ;;next = (pte_t ∗) next_virt_addr;
17 ;;clean up the stack and return next
```

Listing 5.3 Converting a physical address of a PTE to a virtual address (w/o instruction pointer or flag updates).

the side condition, (*), formalizes that any access to the entry with these resources as *invalid*, until the entry is revealed to shared accesses when the presence bit is set.

5.1.4 Physical-to-Virtual Conversion with P2V

Once we reach to the certain knowledge of having an entry with a frame referencing to an allocated resource resides inside the identity mappings (which can already be known if the branch at Line 22 is taken, or ensured by allocating and installing a new entry as just discussed for Lines 22–45), we can utilize this knowledge to convert this frame address into an virtual address of the next page table through, again, identity mappings – Line 57 in Figure 5.1 specified in Figure 5.3.

This actually constitutes a very critical piece of the full page table walk verification, and we have verified the critical step for a small x86-64 kernel, which is the physical-to-virtual conversion (often appearing as a macro `P2V` in C source code). In our small kernel (Line 7 in Figure 5.3), as in larger kernels, `P2V` is actually just addition by the constant offset mentioned in Section 5.1.2, but the correctness of this simple instruction is quite subtle and relies on the extended invariant (Figure 5.1) explained in that section.

Figure 5.3 shows the verification of the end of pte_get_next_table specialized to the case where where

no allocation was necessary (i.e., the conditional on Line 21 of Figure 5.1 was taken). In this case, the present bit being set grants access to the child tokens from Line 6 of Figure 5.1, which is then refined to the assertion on Line 6 of Figure 5.3. The code loads rcx with the offset value KERNBASE, which gives us the value of the virtual address ($\mathsf{entry}_{\mathsf{pfn}}$ +KERNBASE) of the *base* of the next level of the page table. While we could now convert this address to a virtual points-to, this is not necessarily the correct thing to do. The caller walkpgdir (discussed next) uses pte_get_next_table to retrieve just the base address, because only the caller knows which entry in the subsequent table will be accessed (it depends on the corresponding bits from the virtual address being translated). So instead we pass back the per-address-space invariant with the identity mapping resources for entry pulled out. It is up to the caller to determine which entry in that table must actually be accessed — by selecting the appropriate index into the 512 ghost map tokens returned in the postcondition, and using the ghost translation and physical location portions of the invariant to assemble a vpte-pointsto.

5.1.5 Walking Page-Table Tree: Calling pte_get_next_table for Each Level

Realizing a software page-table walk amounts to calling pte_get_next_table for each level as shown in Figure 5.4. The special part of the specficication for a page table walk can be considered as accumulation of memory mappings for the page-table entries visited and frame addresses for page-tables. For example, Lines 29 and 30 in Figure 5.4 show the virtual pte-pointsto assertions for L4 and L3 entries. In the final post-condition, we expect the accumulation of these resources from each level – $\mathsf{R}_{\mathsf{walk}}$ – which allows us to construct and return the path to the L1 entry in the tree to insert a new page.

This code performs most actual physical-to-virtual conversions using the identity mapping portion of the per-address-space invariant. walkpgdir accepts a *virtual* pointer to the base of the L4 table, and the address to translate. The precondition provides knowledge that the virtual base of the L4 is at the appropriate offset from the current cr3 value, but does not provide a virtual points-to assertion because the function must calculate (Lines 5–11) which entry it needs access to. Instead the precondition has 512 identity map tokens, guaranteeing that every entry on the page is subject to

```
;;pte_t *walkpgdir(pte_t *pml4, const void *va) {
... ;; Stack setup
{P * ⌜rtv+KERNBASE = pml4⌝ * ∀_{i∈0..511}(rtv + i * 8) ↪^id 4 * 𝓘ASpace_id(θ, Ξ, m) * rtv ↪^{δs} δ}_rtv
;;set up the stack for root address and virtual address
;;pte_t *pml4_entry = &pml4[PML4EX(va)]; // Virtual address of L4 entry
mov −0x8[rbp],rsi
mov −0x10[rbp],rdi
shr 0x27,rdi ;; Shift L4 index to lowest bits
and 0x1ff,rdi ;; Mask to just lower 9 bits (0x1ff=511)
shl 0x3,rdi ;; Multiply by 8
add rdi,rsi ;; Add to pml4 (virtual) table base
mov rsi,−0x18[rbp] ;; Store to local variable pml4_entry; logical pml4_entry is physical, program variable is virtual
{P * 𝓘ASpace_id(θ, Ξ \ {pml4_entry}, m) * ∀_{i∈0..511\{PML4EX(va)}}(rtv + i * 8) ↪^id 4 * rtv ↪^{δs} δ}_rtv
{entry_present(l4e_val) −* ∀_{i∈0..511} table_root(l4e_val)+ i * 8 ↪^id 3}_rtv
{pml4_entry+KERNBASE ↦_{vpte,qfrac} pml4_entry l4e_val * ⌜qfrac = 1 ↔ ¬(entry_present l4e_val)⌝}_rtv
;;pte_t *pdp = pte_get_next_table(pml4_entry);
mov −0x18[rbp],rdi
...
callq c0 <pte_get_next_table>
;;save the physical next table address in rax
{P * 𝓘ASpace_id(θ, Ξ \ {pml4_entry}, m) * rtv ↪^{δs} δ}_rtv
{∀_{i∈0..511}table_root(l4e_val'.pfn) + i * 8 ↪^id 3 * ⌜pdp-KERNBASE = table_root(l4e_val'.pfn)⌝}
{pml4_entry+KERNBASE ↦_{vpte,qfrac} pml4_entry l4e_val' * ⌜qfrac = 1 ↔ ¬(entry_present l4e_val)⌝}_rtv
{rax ↦_r table_root(l4e_val'.pfn)}_rtv ;; pte_get_next_table may have allocated a new page, updating entry
;;pte_t *pdp_entry = &pdp[PDPEX(va)]; // Virtual address of L3 entry
{⌜pdp + PDPEX(va)*8 = pdp_entry ∧ table_root(l4e_val'.pfn) = pdp⌝}_rtv
{P * 𝓘ASpace_id(θ, Ξ \ {pml4_entry, pdp_entry}, m) * rtv ↪^{δs} δ}_rtv
{∀_{i∈0..511\{PDPEX(va)}} table_root(pml4_entry_pfn) + i * 8 ↦_id 3}_rtv
{pml4_entry+KERNBASE ↦_{vpte,qfrac4} pml4_entry l4e_val' * ⌜qfrac4 = 1 ↔ ¬(entry_present l4e_val')⌝}_rtv
{pdp_entry+KERNBASE ↦_{vpte,qfrac3} pdp_entry l3e_val * ⌜qfrac3 = 1 ↔ ¬(entry_present l3e_val)⌝}_rtv
{entry_present(l3e_val) −* ∀_{i∈0..511} table_root(pdp_entry.pfn) + i * 8 ↪^id 2}_rtv
;;pte_t *pd = pte_get_next_table(pdp_entry);
... ;; Similar assembly to reach next level
;;pte_t *pd_entry = &pd[PDEX(va)]; // Virtual address of L2 entry
;;pte_t *pt = pte_get_next_table(pd_entry);
{R_walk}_rtv
;;access and return L1 entry
;;return &pt[PTEX(va)]; // Virtual address of L1 entry
...
;; clean up the stack
```

Listing 5.4 Walking page-table directory via calls to pte_get_next_table in Listing 5.1

the identity mapping invariant. Line 11 calculates the virtual address of the relevant entry, and the subsequent view shift pulls that entry out of the identity mapping (Ξ) and fetches its corresponding resources as described by Figure 5.1 and Section 5.1.2. The ghost translation and physical location are used to form the virtual pte-pointsto for the L4 entry (Line 23), with the entry validity and next-level indexing satisfying the rest of the precondition for pte_get_next_table. pte_get_next_table then, as described earlier, checks the valid bit in the indicated entry and either returns the (unconditional) tokens for the L3 entry physical addresses (if valid), or allocates into the entry and returns new (also unconditional) tokens for the L3 entry physical addresses. pte_get_next_table's first call (Line 19) returns the virtual address of the base of the L3 table (a *page directory pointer*, so PDP, in official x86-64 terminology). Then the situation to move from that pointer to the base of the L2 is just like the process just followed: the proof calculates the address of the relevant L3 entry, uses the appropriate L3 identity mapping token to construct a virtual pte-pointsto to that entry, and passes that along with additional resources pulled out of the invariant to another call to pte_get_next_table. That call then returns the base of an L2 table, and the process repeats until the function returns the virtual address of the relevant L1 entry. That will then be used in the next section by the caller of walkpgdir to install a new mapping.

5.2 Mapping a New Page

One of the key tasks of a page fault handler in a general-purpose OS kernel is to map new pages into an address space by writing into an existing page table via a call

```
vaspace_mappage(pte_t *pml4, void *va,uintptr_t fpaddr )
```

in Listing 5.5. To do so, with a given allocated fresh page (fpaddr), then we need to locate the appropriate location for the page insertion. We call for a page table walk (via walkpgdir Lines 16-36 Listing 5.4), and update the appropriate L1 page table entry (Line 10 in Listing 5.5).

In Figure 5.5, we see an address (va) currently not mapped to a page (θ !! va = None). Mapping a

Listing 5.5 Specification and proof of code for mapping a new page with $\mathsf{R}_{\mathsf{walk\_acc}}$ in Figure 5.4 expanded.

fresh phyiscal page to back the desired virtual page first requires ensuring the existence of a memory location for an appropriate L1 table entry. The code uses a helper function walkpgdir (discussed again in Section 5.1). walkpgdir's postcondition contains virtual *PTE* pointsto assertions ($\mapsto_{\mathsf{vpte}}$) both for ensuring partial page table walk reaching the L1 entry (pt_entry in Listing 5.5) with the justification of higher levels of the page table exist, and for allowing access to the memory of the L1 entry via virtual address.

The crucial and the only step in addition to traversing the page table in Figure 5.4 is actually updating the L1 entry (Line 11 in Figure 5.5), via the virtual address (pt_entry+KERNBASE) known to translate to the appropriate physical address, in our example the L1 table entry address (addr_to_pfn(pa)).

At this point in the proof, i.e. after the traversal to obtain the sound physical path resources ($\mathsf{R}_{\mathsf{walk\_acc}}$ Line 36 in Listing 5.4), which contains virtual PTE points-to relations along the path to the ownership of the L1 table entry's backing physical memory, whose full ownership is obtained, as we do for the allocation of the missing table entry along the path in Listing 5.4, via exploiting the knowledge of entry-existence as shown in Line 6 Listing 5.5. If there is no entry then we can insert the allocated page (addressed with fpadr in Listing 5.5) by updating the L1 entry to point to the fresh page. Once we insert the fresh page, the whole path reaching the fresh page is complete, and, without violating soundness, we can pull together the physical page-table entry assertions (reside in $\mathcal{R}_{\mathsf{walk\_acc}}$) that are needed for the invariant and construct the virtual points-to assertion for the virtual address ($\mathsf{va} \mapsto_{\mathsf{v,q}} \mathsf{pa}$)

Unlike the only prior work verifying analogous code for mapping a new page[97,98], our proof above does *not* need to reason directly over the operational semantics, making this the first verification we know of for mapping a virtual memory page that stays entirely at the program logic level.

5.2.1 Unmapping a Page

The reverse operation, unmapping a designated page that is currently mapped, would essentially be the reverse of the reasoning around line 22 above: given the virtual points-to assertions for all

512 machine words of memory that the L1 entry would map, and information about the physical location, full permission on the L1 entry could be obtained, allowing the construction of a full virtual PTE pointer for it, setting to 0, and reclaiming the now-unmapped physical memory.

5.3 Change of Address Space

A critical piece of *trusted* code in current verified OS kernels is the assembly code to change the current address space; current verified OS kernels currently lack effective ways to specify and reason about this low-level operation, for reasons outlined in Section 2.

Figure 5.6 gives simplified code for a basic task switch, the heart of an OS scheduler implementation. This is code that saves the context (registers and stack) of the running thread (here in a structure pointed to by rdi's value shown in Lines 5–7 and Line 11 of Figure 5.6) and restores the context of an existing thread (from rsi shown in abbreviated Lines 14–17 and Line 22), including the corresponding change of address space for a target thread in another process. This code assumes the System V AMD64 ABI calling convention, where the normal registers not mentioned are caller-save, and therefore saved on the stack of the thread that calls this code, as well as on the new stack of the thread that is restored, thus only the callee-save registers and `cr3` must be restored.[4] With the addition of a return instruction, this code would satisfy the C function signature[5]

```
1 void swtch(context_t* save, context_t* restore);
```

A call to this code begins executing one thread (up through Line 17) in one address space (rtv), whose information will be saved in a structure at address *old*, and finishes execution executing a different thread in a different address space (whose information is initially in *new*).

Because this code does not directly update the instruction pointer, it is worth explaining *how* this switches threads: by switching address spaces and stacks. This is meant to be called with a return

[4] We are simplifying in a couple basic ways. First, we are ignoring non-integer registers (e.g., floating point, vector registers) entirely. Second, we are ignoring that the caller-save registers should still be initialized to 0 to avoid leaking information across processes. We focus on the core logical requirements.

[5] The name comes from the UNIX 6th Edition swtch function, the source of the infamous "You are not expected to understand this" comment [117].

```
;; Assume the save−space is in rdi, load−space in rsi. First, save the yielding context
{P ∗ IASpace(θ, Ξ, m) ∗ [rtv′](IASpace(θ′, Ξ′, m′) ∗ Pother) ∗ rsi ↦r new ∗ rdi ↦r old ∗ rbx ↦r rbxv}rtv
{rsp ↦r rspv ∗ rbp ↦r rbpv ∗ r12 ↦r r12v ∗ r13 ↦r r13v ∗ r14 ↦r r14v ∗ r15 ↦r r15v}rtv
{ContextAt(old, _) ∗ ContextAt(new, [rbxv′, . . . , rtv′])}rtv
mov 0[rdi], rbx
... ;; mov rsp, rbp, r12, r13, r14, saved to offsets 8, 16, 24, 32, and 40 from rdi
mov 48[rdi], r15
{P ∗ IASpace(θ, Ξ, m) ∗ [rtv′](IASpace(θ′, Ξ′, m′) ∗ Pother) ∗ rsi ↦r new ∗ rdi ↦r old ∗ rbx ↦r rbxv}rtv
{rsp ↦r rspv ∗ rbp ↦r rbpv ∗ r12 ↦r r12v ∗ r13 ↦r r13v ∗ r14 ↦r r14v ∗ r15 ↦r r15v}rtv
{ContextAt(old, [rbxv, . . . , rtv]) ∗ ContextAt(new, [rbxv′, . . . , rtv′])}rtv
mov 56[%rdi], %cr3
{. . . ∗ rdi+56 ↦v rtv}rtv
;; Restore target context
mov rbx, 0[rsi]
mov rsp, 8[rsi] ;; Switch to new stack, which may not be mapped in the current address space!
... ;; load rbp, r12, r13, r14, from offsets 16, 24, 32, and 40 from rsi
mov r15, 48[rsi]
{P ∗ IASpace(θ, Ξ, m) ∗ [rtv′](IASpace(θ′, Ξ′, m′) ∗ Pother) ∗ rsi ↦r new ∗ rdi ↦r old ∗ rbx ↦r rbxv′}rtv
{rsp ↦r rspv′ ∗ rbp ↦r rbpv′ ∗ r12 ↦r r12v′ ∗ r13 ↦r r13v′ ∗ r14 ↦r r14v′ ∗ r15 ↦r r15v′}rtv
{ContextAt(old, [rbxv, . . . , rtv]) ∗ ContextAt(new, [rbxv′, . . . , rtv′])}rtv
;; Switch to the new address space
mov cr3, 56[rsi]
{[rtv](P ∗ IASpace(θ, Ξ, m) ∗ ContextAt(old, [rbxv, . . . , rtv]) ∗ ContextAt(new, [rbxv′, . . . , rtv′]))}rtv
{IASpace(θ′, Ξ′, m′) ∗ Pother ∗ rsi ↦r new ∗ rdi ↦r old ∗ . . .}rtv′
```

Listing 5.6 Basic task switch code that switches address spaces.

address for the current thread stored on the current stack when called — which must be reflected in the calling convention. In particular, the precondition of the return address on the initial stack requires the callee-save register values at the time of the call: those stored in the first half of the code. Likewise, part of the invariant of the stack of the second thread, the one being restored, is that the return address on *that* stack requires the saved callee-save registers stored in that context to be in registers as its precondition.

The wrinkle, and the importance of the modal treatment of assertions, is that the target thread's precondition is *relative to its address space*, not the address space of the calling thread shown as

$$[\mathsf{rtv}'](\mathcal{I}\mathtt{ASpace}(\theta, \Xi, m) * \mathtt{Pother})$$

in the specfication. Thus the precondition of this code, in context, would include that the initial stack pointer (before rsp is updated) has a return address expecting the then-current callee-save register values and suitably updated (i.e., post-return) stack in the *current* (initial) address space; this would be part of P in the precondition. The specification also requires that the stack pointer

saved in the context to restore expects the same of the saved registers and stack *in the other address space*. The other-space modality plays a critical role here; Pother would contain these assumptions in the other address space.

The postcondition is analagous to the precondition, but interpreted *in the new address space*: the then-current (updated) stack would have a return address expecting the new (restored) register values (again, in Pother), and the saved context's invariant captures the precondition for restoring its execution *in the previous address space* (as part of P).

Note that immediately after the page table switch, the points-to information about the saved and restored contexts is guarded by a modality for the retiring address space rtv(Line 23). This is enforced by WriteToRegCtlFromRegModal (Figure 4.4), and is sensible because there is no general guarantee that the data structures of the previous addres space are mapped in the new address space. The ability to transfer that points-to information out of that modality is specific to a given kernel's design. Kernels that map kernel memory into all address spaces would need to ensure and specify enough specific details about memory mappings to allow a proof of an elimination rule for specific modally-constrained points-to assertions.

CHAPTER 6

IMPLEMENTATION

This section gives an overview of the quantification of the contributions made in this part of the thesis, i.e., giving numbers on the gray boxes shown in Figure 1.1.

6.1 Numbers on pte Library

In this section, we would like to quantify the verification effort on the essential virtual-memory functionality of our kernel. The mechanics of this effort start with compiling pte.c and dumping the pte.s. Then, we manually replace the instructions with the ones we explained in the previous sections of this part of the thesis. Please note that the numbers referring to proof efforts should be taken approximate numbers because this is still a developping framework, and refinements in the proof code base constantly occur.

In Table 6.1, we see the approximate numbers for the functions we verified from the page-table library (pte_get_next_table, pte_walkpgdir, pte_initialize, and pte_map_page), an address space switch task referring some virtual points-to resources (pte_switch_addrspace), and single instruction w that semantically exhibit challenging proofs (pte_p2v).

Table 6.1 Line-of-Code Numbers for pte Verification

	C LoC A	Assembly LoC	Roqc Proof LoC
pte_get_next_table	12	45	3200
pte_walkpgdir	8	44	3200
pte_p2v	–	1	75
pte_switch_addrspace	–	18	350
pte_map_page	7	28	1750
pte_initialize	4	20	700

6.2 Numbers on x64-Iris

The logical machinery that enables reasoning for pte library is x64-Iris. In Table 6.2, we give the Roqc number of implementation lines for our machine model. Another line mentions the number of proof lines for the instructions mentioned in this thesis. To shed a light on the number of proof lines in the complete set of instructions excluding interrupt handling, our non-modular proof attempt, that is, not separating the page-table-traversal into another lemma to be applied in the proof of instructions accessing memory, ended up being at least more than 25 times of the number mentioned in Table 6.2. As a final remark, we also give the number of lines for the definitions and lemmas for x64-Iris which includes definitions of only VMM related constructions (e.g., address-space modality and virtual pointsto). In other words, this number excludes the extra Iris construction that helps in performing Iris proofs.

Table 6.2 Line-of-Code Numbers for x64-Iris Logic

	Roqc LoC
Soundness of Instructions Mentioned in the Thesis	50176
VMM Related Logical Constructions	5554
Machine Model	6172

CHAPTER 7

CONCLUSIONS

This thesis advances the state of the art in formal verification of programs subject to address translation on hardware using virtual memory. We proposed to treat assertions about virtual memory locations explicitly as assertions in a modal logic, where the notion of context is the choice of current address space based on the page table root installed on the CPU. We improved the modularity of our virtual address translation to allow page table modifications that preserve mappings without collecting all affected virtual points-to assertions. To make specifications of code involving other address spaces cleaner, we borrowed the idea of modalities which explicitly name the conditions under which they are true from hybrid logics. We implemented these ideas in a derived separation logic within Iris, and proved soundness of the rules for essential memory- and address-space-change-related x86-64 instructions sound against a hardware model of 64-bit 4-level address translation. Finally, we used our rules to verify the correctness of key VMM instruction sequences, including giving the first assembly-level proof of correctness for a change of address space expressing which assertions hold in which address space, and the first physical-to-virtual translation proof.

Part III

Modal Understanding of Modularity of State-Transition-Systems

CHAPTER 1

BACKGROUND

An important element of verifying stateful computer systems is specifying the allowed orders of operations on a data structure, subsystem, or remote system, along with how these operations update the state and interfere with requests from concurrent clients of the same resource. The past few years have seen *state transition systems* (STSes) gain popularityfor this purpose, particularly for specifying protocols for threads to cooperate on modification of shared state[53,91,139,156,157]. An STS is specified by giving a set of abstract states, each with an invariant over the structure for that abstract state (typically, an invariant in a rich separation logic) and a set of *tokens* owned by the structure in that state, along with a transition relation specifying how the structure may evolve between abstract states. To verify that a mutation moves the data structure along the protocol correctly, each update from a given abstract state must modify the program state to match a permitted destination state's invariant, including token ownership[1].

Crucially the program at the point of mutation must own any tokens required by the final abstract state, and afterwards is considered to own any tokens previously owned by the initial state (and not also owned by the final state). This token exchange forms the basis of permission exchange between threads — a thread may only induce a transition if it owns the required tokens, which may be exchanged by interacting through the data structure. For example, modeling a basic mutual exclusion lock in this way includes two abstract states **locked** and **unlocked**, and two tokens lock and unlock. In the **locked** state, the data structure's representation includes ownership of the token

[1]We are simplifying slightly; the knowledge of the origin and destination may be imprecise, and therefore this check is done for *each possible* initial and final abstract state.

unlock; unlocking the structure (releasing the lock) requires a state update to match the invariant of **unlocked**, which requires transfering the unlock token into the data structure, were a subsequent lock acquisition will obtain it.

This class of specifications is flexible, intuitively appealing, and with a modest extension, permits adapting the long-standing idea of specifying a system with a state machine to support concurrent clients: token ownership may be fractional (for both the client code and the auxilliary state of the data structure itself), which induces a natural variant of rely-guarantee reasoning[87]: a client is limited to those transitions enabled with the tokens it owns, and interference from concurrent clients is limited to at most those transitions made possible by the *complement* of the client's token ownership (i.e., assuming there is a single opposing client posessing all tokens the local client does not).

STSes offer a concise, high-level specification for the interactions between concurrent clients of a data structure, subsystem, or server, and those communications' effects on data structures. However, modularity for STSes has not been thoroughly studied. Early systems lack modular STSes[156] beyond simple nesting (e.g., the invariant for one STS state referring to ownership of another STS's state), achieve limited forms as a consequence of working with impredicative higher-order separation logics (which offer limited forms of qualification, and therefore subsumption)[150], or are limited to essentially a product construction over STSes where code verified against one component can be used with a product containing it[144] (this last is a simplification we will revisit in detail later).

This is a problem, because real systems are rife with implicit protocols, but with differences between them that current approaches to modular STSes are ill-suited to support. Consider, for example, the various layers of filesystem abstraction in an operating system kernel. The kernel specifies a range of operations on files, but most application code uses only a small subset of them (opening, reading, writing, and closing). This is especially important for cross-platform code: different kernels offer different operations (and some offer different semantics for common operations!) so applications code to the common (consistent) subset. Supporting such verification requires being able to abstract

kernel-exposed file protocols in such a way that clients can ignore certain operations and file states. At the same time, the kernel requires the lower level filesystem drivers to implement a range of operations sufficient for the kernel to implement the full range of operations it dictates (even if client programs ignore many of them). However, some operations make no sense for some filesystem. If we go back to our example the in-memory filesystem backed by a chunk of RAM: there is no sensible notion of syncing such a filesystem with the disk, because by design it is not backed by disk. So to verify that an in-memory filesystem adheres to the protocol required by the kernel — which would specify separate abstract states for synced and un-synced data — we would need to abstract the protocol actually obeyed by the in-memory filesystem in such a way that it adds additional states and transitions that map onto the ones that actually exist from the filesystem's perspective. Aside from ad-hoc means, current approaches do not support both kinds of abstraction, and those that support one kind use mechanisms where the sorts of permissible abstraction must be planned for up front.

In this thesis, we propose a single form of abstraction for STS specifications that supports both hiding states and operations (as with an application ignoring operations the kernel permits) and fabricating states and operations (as in fitting the in-memory filesystem's real operations into a more permissive driver protocol). We extend and adapt the classic notion of bisimulation (in the modal logic sense) by treating STSes as essentially a specialized form of Kripke structure; these notions and a proof rule that permits this form of "subtyping" on STSes are formalized as an extension to the Iris[92] formalization of protocols. We then demonstrate how this works for both concurrent data structures and examples from standard abstraction layers in operating system kernels.

1.1 A Primer on Concurrent Program Logics

Early Concurrent Program Logics Concurrent program logics have a long history, stretching back over 40 years now to the original Owicki-Gries logic[135]. Owicki and Gries were the first to clearly articulate that one way to ensure safe composition of individual threads' proofs was to check that when two threads were composed in parallel, no action of one thread could violate an assumption made in the proof of the other. This of course suffered from requiring a quadratic number of checks (each statement of each thread must be checked against each invariant of each other thread), but the recognition that thread actions mustn't interfere with non-local *proofs* established the key idea used in modern logics today. Jones[87] introduced *rely-guarantee* reasoning: rather than checking each statement against any possibly-conflicting assertion, he abstracted: each thread was checked with a relation giving an upper bound on any individual action of any other thread (the rely) and each local assertion was checked to be preserved by that relation (i.e., *stable* w.r.t. that relation). Each thread was also checked with a guarantee: an upper bound on each individual local action. Then parallel composition could be validated by comparing the guarantee of one thread to the rely of another. This made checking thread-modular (to a point), and the number of checks was now again linear in the size of the program.

Both these and other early concurrent program logics suffered from the same state modularity issues as traditional Hoare logics: assertions were (possibly) global, making reuse across programs difficult. Separation logic[133,140] mostly solved this state modularity issue in the sequential case, and Brookes was the first to show how to extend separation logic with threads and Hoare-style critical sections with (spatial) invariants[23].

Brookes' Concurrent Separation Logic (CSL)[23] permitted the use of separation logic for concurrent programs, but by enforcing the strong non-interference property inherent in any lock-based data-race-free system.

Abundance of Concurrent Program Logics Rely-guarantee offered more granular interference, but lacked the modularity and strong reasoning about heap updates present in separation logic. So combining them was a natural next step. Vafeiadis and Parkinson introduced a logic RGSep[158], which integrated the two. Each thread had both a local heap described by a separation logic assertion, and knowledge of shared region described by an assertion that was stable with respect to a pair of relational (two-state) specifications of the sorts of modifications that were possible to the shared state: a rely and guarantee, though adapted to use separation logic to give the conditions. This approach to retaining the local reasoning power of separation logic while permitting fine-grained concurrency (not possible using only Hoare-style monitor locks as in the original concurrent separation logic work) kicked off a flurry of activity on logics marrying the two. Feng et al.[61] explored a way to formulate an RGSep-style logic as a refinement of traditional rely-guarantee reasoning. Local rely-gaurantee[60] defined separating conjunction on the rely/guarantee relations themselves to decompose interference.

Generalization This first round of new logics was sometimes viewed as awkward because the rely/guarantee relations extended Hoare triples with additional components that had to be plumbed through all parts of the proof — even portions that were mostly thread-local. So work started to explore embedding the elements of interference control as resources within separation logic assertions. Deny-guarantee[55] incorporated the notion of not only describing what permissions were possibly-granted to interfering threads, but also what permissions were explicitly denied to other threads. Concurrent abstract predicates (CAP)[53] brought the notion of a first-class specification of interference over a region of memory — where regions were explicitly identified in assertions and labeled with a protocol describing interference — to the fore. That work also introduced the use of *tokens*: bits of ghost state that could be fractionally split between threads and (ghost) state, making it more straightforward to encode sharing or transferrence of rights (in the deny-guarantee sense) to perform certain updates. This was later followed by an impredicative extension that significantly increased its power[149].

Following the development of CAP, explicit protocol definitions over explicit regions of memory

became a recurring theme, appearing in CaReSL[156], FCSL[144], Iris[91], and CoLoSL[139] in various forms.

iCAP is actually a little unusual. It uses the power of it's higher order specification logic to qualify some hoare triples. But there's no notion or examples of weakening a spec given a weaker interference guarantee. It does use tokens, but the transition system (as in CAP) is implicit in the triples exported by a module.

Tokens, State Transition Systems & Interference This thesis utilizes the logical state transition system constructions grounded in CaReSL[156]. Protocols in CaReSL[156] are logical abstractions with the states, transition relations enabling taking steps in between any two steps, and interpretation function which enables what *abstractly* protocols conveys concretely per state. An example protocol from CaReSL[156] to govern a lock would be composed of two states: Unlocked and Locked. As expected a lock's state can be changed from being locked to unlock and vice versa, therefore, there would be transition relations in both ways. Intuitively speaking, we expect at most one thread to obtain the lock, i.e. lock state changes from Unlocked to Locked. CaReSL enables coordiantion of threads via *token* ownership. Speaking concretely, for our example protocol, the protocol owns a token called TokLock at the protocol state UnLocked, and for a thread to take a step from Unlocked to Locked, it needs to *earn* (take the ownership of) the token TokLock from the protocol. Conversely, to release a lock, i.e. to take a step from Locked state to Unlocked state,a thread needs to *pay* (giving up the ownership of) the previously earned token TokLock [156]. Although, we discuss it in the further chapters of this part of the thesis, at the high-level, constitutes the foundation of *token-sensitive transition relations*: a transition from a state with a set of token tokens (s,T) to one another state with another set of tokens (s,T') is permitted as long as the *the disjoint union of the tokens owned by the thread and the protocol-owned tokens are preserved before and after the step is taken* – the law of token conservation [156].

$$T \biguplus \mathcal{T}(s) \equiv T' \biguplus \mathcal{T}(s)'$$

Based on the token-sensitive transitions, CaReSL[156] identifies what permissible transitions for a thread to take ($\overset{\mathsf{guar.}}{\sqsubseteq}_\pi$ transitions enabled by the thread owned tokens) and for the environment (other threads) to take ($\overset{\mathsf{rely}}{\sqsubseteq}_\pi$ transitions enabled by the tokens owned by other threads).

As a final remark on the interference, we should also note that, unlike CaReSL[156], Views[51] asks for the interference relation of partial commutative monoids as an input to the logic dictating how two interference resources can be composed.

CHAPTER 2

PROTOCOLS

2.1 Encoding Protocols in STSes

The past few years have seen the rise of *state transition systems* (STSes) for specifying protocols over shared state. An STS π in this context (following CaReSL's presentation [156]) is given by:

1. a set of states $\mathcal{S}$,
2. a map from a state set of tokens $\mathcal{T} : \mathcal{S} \to \mathsf{TokSet}$,
3. a transition relation $\rightsquigarrow$ on states, which is then lifted to pairs of a state and token set:

$$(s; T) \rightsquigarrow (s'; T') \stackrel{\text{def}}{=} s \rightsquigarrow s' \land \mathcal{T}(s) \uplus T = \mathcal{T}(s') \uplus T'$$

4. an interpretation mapping states to state assertions $\phi : \mathcal{S} \to \mathsf{Prop}$.

Logics incorporate a notion of a region n governed by a protocol π, and an assertion that the region is in *at least* particular state with a particular set of tokens owned by the current view:

$$\boxed{s; T}^{n}_{\pi} \approx \exists s'.\, s \rightsquigarrow^{*} s' \land \phi(s') \land \mathsf{RegionOwned}(n, \mathcal{T}(s')) \land \mathsf{LocalToks}(T)$$

These regions are referred to as *islands*, and the locally-owned token set induces a *rely* and *guarantee* (in the sense of rely-guarantee reasoning [87]) that bound interference possible from other parts of the program, and actions available with only the given assertion. Protocol states advance according to

an island update rule (UpdIsl), *approximately*[1]:

$$\textsc{UpdIsl}\quad \frac{\forall b_0.\, b \overset{\mathsf{rely}}{\sqsubseteq}_\pi b_0 \to \{\pi.\phi(b_0) * P\}\ \alpha\ \{\exists b'.\, b_0 \overset{\mathsf{guar}}{\sqsubseteq}_\pi b' * \pi.\phi(b') * Q\} \qquad \alpha \text{ physically atomic}}{\{\boxed{b}^n_\pi * P\}\ \alpha\ \{\exists b'.\, \boxed{b'}^n_\pi * Q\}}$$

This rule explicitly quantifies over all possible protocol states other threads (parts of the program) may have moved the state to, and ensures that the behavior of atomic action α is valid in all such states.

Soundness of UpdIsl in Iris While STSes were formalized with the initial release of Iris, the rule was never actually proven sound. We present the first actual proof of this rule within Iris as a small additional contribution of the thesis.

2.2 Limitations of Existing STS Logics

The initial formulations of logics with STSes for concurrency — both (i)CAP-style and CaReSL-style (as retained by Iris) — lack modularity beyond simple nesting of STS-based data structures. In iCAP (later found to be unsound) and Iris (the fix), some modularity arises from the use of a higher-order separation logic, which permits qualifying the use of some protocols, and defining protocols in terms of each other, but this is not due to any special handling of STSes.

One form of compositionality is the ability to *compose* STSes. Nanevski et al.[129] and Sergey et al.[144] introduced a very different style of STS, with explicit input and output channels for exchanging state with other STSes, which in conjunction with the use of subjective auxilliary state[113], permits an expressive *entanglement* operation for joining two STSes into a single larger system, along with a form of framing that permits operations defined on a given STS to operate on entanglements containing that STS. This offers an elegant, but incomplete form of modularity: larger STSes can be constructed and original operations reused without re-verifying, but the the granularity of

[1] We intentionally omit some details from the presentation above related to a later modality in the logic to internalize step indexing in the semantics of CaReSL.

composition is fixed to entangling (implicitly) named STSes. Semantically-compatible operations verified against syntactically unrelated STSes cannot be reused across structures.

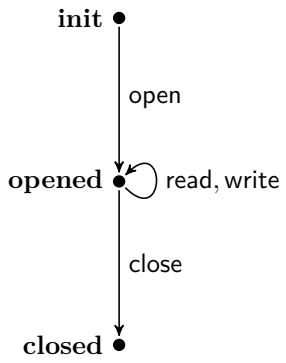
Another take on modularity is the ability to view protocol-governed state as if governed by a smaller protocol with fewer transitions, over a subset of the relevant state. Raad et al.[139] enrich an iCAP-style protocol logic with such a notion of modularity, allowing a thread to locally forget about actions it does not require and state it will not modify, as long as the resulting smaller-footprint view does not become unstable with respect to the "ground truth" protocol for a region.

Notably missing from these explorations of modularity in STS logics is a form of *semantic decomposition*: the ability to use an operation that is verified against one STS with a different STS, for which the operation is semantically compatible (under some conditions). This follows for free in traditional rely-guarantee logics: the traditional rely-guarantee rule of consequence permits using a more permissive rely and less permissive guarantee relation, judged semantically by checking relation containment. Some descendants also inherit forms of this, such as Local Rely-Guarantee[60]'s notion of framing on relations. Raad et al.'s work is a form of semantic decomposition, but done in a setting that is already characterized only semantically (with CAP-style protocols).

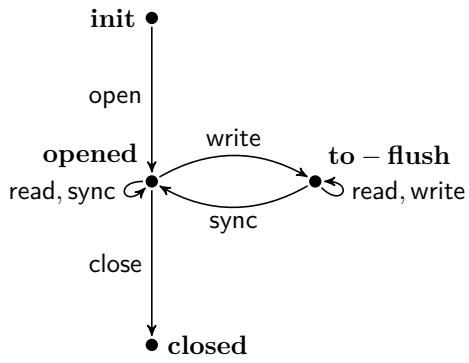
Semantic Decomposition In this thesis, we provide a form of semantic decomposition for protocol-based STSes.

2.3 Intuition Behind "Subtyping" STSes

We build on the notion of bisimulation between Kripke models to induce a stronger rule of consequence. In particular, we use the restriction of an STS π to a subset of its capabilities (or to the states reachable from a given state and token set) to induce an STS-specific form of generated submodel. An STS-appropriate bisimulation between this submodel and another STS π' allows the use of code verified against π' when the context can ensure (by framing tokens) that portions of π not modeled by π' remain inaccessible.



(a) Traditional file I/O protocol



(b) A possible distributed file I/O protocol

Figure 2.1 File I/O protocols

2.3.1 Motivation

Consider a STS for a protocol that enforces the usage of a file resource. Figure 2.1a shows a typical model of correct usage of a file resource: it initially exists, then can be opened, used arbitrarily many times (for reading or writing), and finally closed (once).

Figure 2.1b shows one possible protocol for a client of a distributed filesystem, where no changes are stored until an explicit synchronization operation is invoked. For expository clarity and brevity, we will assume there is only a single client of this remote storage system, and that changes in the traditional protocol are immediately written to disk; both are significant simplifications compared to real systems, but illustrate the important points.

$$\varphi_{\mathsf{distributedfile}}(\ \ell\ ,\ \mathsf{R}\ )(\ s\ ) \triangleq \left\{ \begin{array}{ll} \mathsf{match\ s\ with} & \\ \mathsf{to-flush} \Rightarrow & \mathsf{R} * \exists\ \mathsf{fs.\ isValidDirty(fs)} * \ell \mapsto (\mathsf{fs}.id, \mathsf{fs.status} = \mathsf{dirty}) \\ \mathsf{opened} \Rightarrow & \mathsf{R} * \exists\ \mathsf{fs.\ isValid(fs)} * \ell \mapsto (\mathsf{fs}.id, \mathsf{clean}) \\ \mathsf{closed} \Rightarrow & \exists\ \mathsf{fs.\ isValidClosed(fs)} * \ell \mapsto (\mathsf{fs}.id, \mathsf{closed}) \end{array} \right\}$$

$\varphi_{distributedfile}$ associates the abstract state s with a simple concrete file state ($\mathsf{fs} \in \mathsf{N} * \mathsf{N}$) mentioned in an invariant-per-state that represents the file content (e.g., isValidClean) and the fact specific to the file's *status* (e.g.,clean). Because it is connecting the physical state to an abstract one used in the

$$\{\boxed{\mathsf{opened}; T}^{n}_{\mathsf{file}}\}\ \mathsf{write}(\ell, \mathsf{data})\ \{\exists \mathsf{T}'\ .\ \boxed{\mathsf{opened}; T'}^{n}_{\mathsf{file}}\}$$

Figure 2.2 A File Library: writing to a file.

$$\{\boxed{\mathsf{opened}; T}^{n}_{\mathsf{file}}\}\ \mathsf{write}(\ell, \mathsf{data})\ \{\exists \mathsf{T}'\ .\ \boxed{\mathsf{opened}; T'}^{n}_{\mathsf{file}}\}$$

$$\rightarrow$$

$$\{\boxed{\mathsf{opened}; T}^{n}_{\mathsf{distributedfile}}\}\ \mathsf{write}(\ell, \mathsf{data})\ \{\exists \mathsf{T}'\ .\ \boxed{\mathsf{opened}; T'}^{n}_{\mathsf{distributedfile}}\}$$

Figure 2.3 Transferring the Proof of a File Library: writing to a file.

specification, defining the state interpretation is the first essential step to check client code follows the specification encoded as state machines, but then we need to define the constructions and rules defining the protocol and orchestrating accessibility to it.

$$\varphi_{\mathsf{file}}\ \ell\ \mathsf{R}\ s\ \triangleq \left\{ \begin{array}{ll} \mathsf{match\ s\ with} & \\ \mathsf{opened} \Rightarrow & \mathsf{R} * \exists\ \mathsf{fs}.\ \mathsf{isValid}(\mathsf{fs}) * \ell \mapsto (\mathsf{fs}.id, \mathsf{fs.status} = \mathsf{clean} \vee \mathsf{dirty}) \\ \mathsf{closed} \Rightarrow & \exists\ \mathsf{fs}.\ \mathsf{isValidClosed}(\mathsf{fs}) * \ell \mapsto (\mathsf{fs}.id, \mathsf{fs.status} = \mathsf{closed}) \end{array} \right\}$$

which is simply $\varphi_{\mathsf{distributedfile}}$ the state to-flush removed.

In Figure 2.2 there is a file library client code writing to a file. Suppose that the client code wants to use a library that has a proof against the traditional STS (Figure 2.1a) which is the specification $\boxed{\mathsf{opened}; T}^{n}_{\mathsf{file}}$ with state interpretation φ_{file} , and shown as the premise of the implication in Figure 2.3. But, we want to use that library proof with a file object following the distributed file STS (Figure 2.1b) which is the specification $\boxed{\mathsf{opened}; T'}^{n}_{\mathsf{distributedfile}}$ with state interpretation $\varphi_{\mathsf{distributedfile}}$, and shown as the goal of the implication in Figure 2.3. The distributed file STS forces you to sync to disk before closing the file. As long as the library code doesn't close the file, it's actually safe to use the client code with this new STS (Figure 2.1b), in other words the states opened and to-flush are *indistinguishable* from the client's perspective. Our goal is to introduce the required reasoning principle to prove the implication in Figure 2.3 so that we can preserve the proof against the state

machine in Figure 2.1a to the state machine in Figure 2.1b.

CHAPTER 3

KRIPKE MODELS, BISIMULATION, AND GENERATED SUBMODELS

Bisimulation is a notion of two (possibly infinite) transition systems having common structure (behavior). It has arisen independently in multiple contexts[143]: computer science, modal logic, and set theory. Each domain has its own slightly different definition, terminology and notation; here we recall the essentials of bisimulation from modal logic, specifically bisimulation of Kripke models.

Kripke models are the most widely-used sort of structure for defining the meaning of propositions in modal logic, corresponding to the well-known possible-world semantics proposed by Kripke[102]. There a modal formula's truth depends on the circumstances under which it is evaluated: a simple temporal example would be that the truth of "It is raining today" depends on the particular day on which the sentence is claimed. Thus an abstract set of *worlds* is chosen, a way is given of knowing which propositions are true in which worlds, and critically a relation on worlds is used to interpret modal operators. More formally:

Definition 1 ((Propositional) Kripke Model[84]) *A Kripke model* $\mathfrak{M}$ *is a triple* (W, R, V) *where*

- W *is a set of "worlds"*
- $R \subseteq W \times W$ *is a relation called the* accessibility *relation between worlds*
- $V : \mathsf{PropVar} \to \mathcal{P}(W)$ *gives for each propositional variable* p *a set of worlds* $V(p)$ *where* p *is considered true*

In the case that there are multiple modal operators to interpret, R is replaced by a set $R_{i \in I}$ of binary

relations on worlds, one for each modal operator. In this case, modal formulae are interpreted with respect to a particular world w (or "state" s), and modal operators are typically interpreted as:

$$\begin{aligned}
\mathfrak{M}, w &\vDash p &&\Leftrightarrow w \in V(p) \\
\mathfrak{M}, w &\vDash [m]P &&\Leftrightarrow \forall w'.\, wR_m w' \Rightarrow \mathfrak{M}, w' \vDash P \\
\mathfrak{M}, w &\vDash \langle m \rangle P &&\Leftrightarrow \exists w'.\, wR_m w' \Rightarrow \mathfrak{M}, w' \vDash P
\end{aligned}$$

The exact meaning of the accessibility relations depends on the purpose of the logic at hand. Because we are interested in computer programs, a natural example is the propositional fragment [62] of dynamic logic [138]. There, the modalities are programs (whether atomic statements, or compound statements), and the proposition $[c]P$ represents the claim that after executing c, P will be true — which should be true only in those worlds (program states) where executing c will in fact make P true, and so is actually the weakest precondition operation. This can in fact be used to build up Hoare triples: $\{P\}C\{Q\}$ can be encoded as $P \Rightarrow [C]Q$. This is actually the approach used by Iris [100] to build Hoare triples from a weakest precondition calculus.

In general it is often the case that two different models are actually indistinguishable in the modal logic: they are mathematically different structures, yet no formulae is true in one and false in the other. Dynamic logic again provides a useful example: in the case of *deterministic* programs, it must be that $[m]P \Leftrightarrow \langle m \rangle P$ (since there should only be one possible result for running a program from a given start state). Clearly this holds when for any w and m, there is exactly one w' such that $wR_m w'$. However, it also holds for a broad class of intuitively similar models: those where instead of requiring that each state have exactly one successor for each relation, we instead require that when a world has multiple sucessor states, they have all the same successors and the same propositional variables are true in each:

$$\forall w, w', w'', m.\, wR_m w' \wedge wR_m w'' \Rightarrow (\forall p.\, w' \in V(p) \Leftrightarrow w'' \in V(p)) \wedge (\forall s, m'.\, w' R_{m'} s \Leftrightarrow w'' R_{m'} s)$$

This intuitive notion of two models being equivalent is captured precisely by the notion of bisimulation:

Definition 2 ((Propositional) Bisimulation of Kripke Structures) *A* bisimulation *between (multimodal) Kripke structures* $(W, R_{i\in I}, V)$ *and* $(W', R'_{i\in I}, V')$ *is a relation* $E \subseteq W \times W'$ *satisfying:*

- *If* $w\ E\ w'$*, then* w *and* w' *satisfy the same propositional variables.*
- *If* $w\ E\ w'$ *and* $w\ Rv$*, then there exists* $v' \in W'$ *such that* $v\ E\ v'$ *and* $w'\ R'\ v'$
- *If* $w\ E\ w'$ *and* $w'\ R'\ v'$*, then there exists* $v \in W$ *such that* $v\ R\ v'$ *and* $w\ R\ v$

Two models $\mathfrak{M}$ *and* $\mathfrak{M}'$ *for which a bisimulation exists are said to be* bisimilar*, written* $\mathfrak{M} \sim \mathfrak{M}'$*.*

The point of defining bisimilarity is to prove that two bisimilar models satisfy the same formulae, though doing so requires a full definition of the formulae under consideration; we defer such discussion until we are ready to prove this for a particular language.

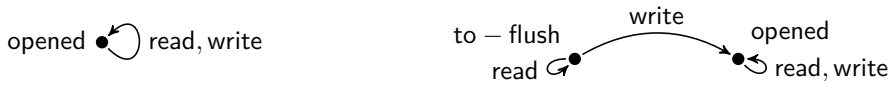
There are many standard constructions on models that give rise to bisimulations. One common one we will adapt is that of a *generated submodel*. Given a Kripke model, a generated submodel corresponds to the portion of the model accessible from a given state.

Definition 3 (Multimodal Generated Submodel) *Given a multimodal Kripke model* $\mathfrak{M} = (W, R_{i\in I}, V)$*, a generated submodel for a subset* $W' \subseteq W$ *and a subset* $I' \subseteq I$ *is* $(W', R'_{i\in I'}, V')$ *if* $\forall i \in I'. R'_i = R_i \cap (W' \times W')$*,* $\forall p. V'(p) = V(p) \cap W'$*, and* W' *is* $R_{I'}$*-closed:* $\forall i \in I', u \in W', v \in W. uR_iv \Rightarrow v \in W'$*. Note the closure only applies to the subset of accessbility relations considered in the submodel.*

We say a generated submodel $(W', R'_{i\in I'}, V')$ *is* generated by *a point*[1] $s \in W'$ *and a subset* $I' \subseteq I$ *if every element of* W' *is reachable from* s *via some finite sequence of steps via the accessibility relations of* $R'_{i\in I'}$ *(i.e., it contains* only *those points reachable from* s *via the selected subset of modalities).*

To illustrate generated submodels and bisimulation, we can consider the usage protocols in Figures 2.1a and 2.1b. In fact, these figures represent multimodal Kripke models. The set W of worlds is the set of points (labeled in boldmathfont), and the accessibility relations are given visually as labelled edges in the graph: an edge $\mathsf{A} \xrightarrow{\mathsf{foo}} \mathsf{B}$ indicates $(\mathsf{A}, \mathsf{B}) \in R_{\mathsf{foo}}$. An edge with multiple labels indicates

[1] This definition is often given in terms of a set of points rather than only the singleton, but we will not need this for our purposes.



(a) Submodel of traditional I/O protocol generated by opened

(b) Submodel of the distributed I/O protocol generated by $\mathsf{to-flush}$

Figure 3.1 Submodels of I/O protocols

that pair of states is in multiple accessibility relations.

Most code making use of files actually does not care about anything but the central read/write loop in the protocol, shown in Figure 3.1a. In most applications, the actual opening and closing of files is isolated to a narrow part of the application, such as the setup and cleanup of a logging subsystem, while most application code is unaware of the backing storage of a file. This *subprotocol* is in fact a generated submodel, generated from opened and the read and write relations. It is bisimilar with the original model (relating opened to itself), as is the case for all generated submodels:

Lemma 1 (Generated Submodels Bisimilarity[19]) *Given a model* $\mathfrak{M} = (W, R_{i\in I}, V)$ *and a generated submodel* $\mathfrak{M}' = (W', R'_{i\in I'}, V')$ *of* $\mathfrak{M}$, $\mathfrak{M} \sim \mathfrak{M}'$.

In all cases, the bisimulation between a model and its generated submodel is always given by the identity relation on the submodel's states.

What does this mean for files? Intuitively, it *should* mean that any code certified to be correct with regard to this generated submodel is safe to use with a file obeying the full protocol of Figure 3.1a, though this depends on the details of how programs are connected to protocols; we will explain the technical details later, but for now note that the notion of protocol with tokens now common in concurrent program logics like CaReSL and Iris is a suitable way to make this connection, and one outcome of the developments later in this thesis is to make precise what this "should" actually means and to show that it holds true in Iris.

Definition 4 (Restricted Submodel) *The* restricted submodel π^t/U *of an STS* π *with respect to a state* t *and a set of (framed) tokens* U *is defined as:*

- *States:* $\mathcal{S}' = \{s \mid s \in \pi.\mathcal{S} \wedge s \to_\pi^* t \wedge \pi.\mathcal{T}(s) \cap U = \emptyset\}$

- *Token map:* $\mathcal{T}|_{\mathcal{S}'}$ *($\pi.\mathcal{T}$ restricted to the states still in the restricted submodel).*
- *Transition relation* $\rightarrow' = \rightarrow|_{\mathcal{S}'}$
- *Interpretation map* $\varphi|_{\mathcal{S}'}$

Intuitively, the definition of the restricted submodel restricts every component of a state machine π to only those states that are accessible (regardless of tokens) from the "current" state t and do not interact with the removed tokens.

Concretely speaking, Figure 3.1b is a generated submodel of Figure 2.1b, generated from the state $\mathsf{to-flush}$ and the same relations. As with the submodel of the traditional file I/O protocol and all generated submodels (Lemma 1) it is bisimilar with the full protocol. But there is an additional wrinkle here: the two generated submodels are *bisimilar to each other*! The bisimulation relation in this case relates opened in the traditional submodel with both $\mathsf{to-flush}$ and opened in the explicit-synchronization submodel. So (in principle) code verified to work correctly using the traditional subprotocol of Figure 3.1a is *also safe to use with files usable with Figure 3.1b*!

This is an abstraction — using client code verified against the (sub)protocol in Figure 3.1a with files obeying the protocol of Figure 2.1b (and in one of the appropriate states) — is impossible with most current approaches to using protocols in concurrent program logics. CaReSL [156] allows using protocols in the state invariants of "outer" protocols (e.g., using a range of smaller protcols in an implementation of flat-combining), but our example is one where code for one protocol is used directly against resources following a semantically-compatible (sub)protocol. FCSL [144] allows a specialized product construction to compose protocols, and then permits code verified against one component to be used with a product containing that component, in a way resembling structural subtyping. While it may superficially appear that the traditional subprotocol is such a building block of the synchronizing subprotocol (both contain a read/write loop on opened), we will not require such relationships to play a role in the definition and construction of protocols (FCSL does), and in practice the representation invariants (not net discussed) of the $\mathsf{to-flush}$ and opened states may be different, and FCSL would not permit that.

Most other work in this style does not consider this kind of subtyping at all. The one exception of a sort would be logics with impredicative quantification (i.e., iCAP[149] and Iris[91]), which in principle allow a specification to quantify over all semantically-compatible inputs. However, this discards one of the primary benefits of protocol-based specifications, replacing the appealing structural specifications with quantification over relations with certain properties. One view of our goals is to harness this power in terms of semantically compatible *other protocols*, which retains the structural flavor of reasoning without sacrificing semantic compatibility entirely.

The rest of this part in the thesis generalizes these ideas to the protocols typically used in modern concurrent program logics and demonstrates that doing so addresses significant issues with modular verification in the presence of protocols. Specifically, we show how to adapt the ideas of multimodal bisimulation and generated submodels to CaReSL/Iris[91,156] style protocols that include representation invariants and token ownership, and how this permits new kinds of abstraction that are useful in verifying common and important software design patterns.

CHAPTER 4

AN ATTEMPT AT STS BISIMULATION

In this section, we start by defining the pieces of our morphism relation and continue by introducing the relations that constitute our bisimulation relation which we define as a single morphism.

4.1 Definitions

Consider two state machines π and π' consist of relations:

- $(\epsilon_{\mathcal{T}} : \mathsf{Tok} \rightharpoonup \mathsf{Tok})$: embeds a token from one state machine to another, e.g. wa of π is embedded in wa of π' – $\mathsf{wa}\ \epsilon_{\mathcal{T}}\ \mathsf{wa}$.
- $(\epsilon_{\mathcal{S}} : \mathcal{S} \to \mathcal{S})$: embeds a state from one state machine to another one, e.g., $\mathsf{to\text{-}flush}$ of π is embedded into opened of π' – $\mathsf{flushed}\ \epsilon_{\mathcal{S}}\ \mathsf{opened}$.
- $(\overset{\mathsf{guar.}}{\sqsubseteq}_{\pi} : \mathsf{Rel}(\mathcal{S} * \mathsf{TokSet}))$ relation defined over a state and a token set such that any local step taken from the state *locally* is made available w.r.t to the tokens used by the *client*, e.g., $(\mathsf{opened}, \emptyset)\ \overset{\mathsf{guar.}}{\sqsubseteq}_{\pi}\ (\mathsf{flushed}, \emptyset))$
- $(\overset{\mathsf{rely}}{\sqsubseteq}_{\pi} : \mathsf{Rel}(\mathcal{S} * \mathsf{TokSet}))$: relation defined over a state and a token set such that any steps taken from the state is made available w.r.t to the token used by the *environment*, e.g., π' can take local steps with $\emptyset$ token set while π can take steps using $\{wa\}$ – $(\mathsf{opened}, \emptyset)\ \overset{\mathsf{rely}}{\sqsubseteq}_{\pi'}\ (\mathsf{opened}, \emptyset))$.

Embedding of Tokens Figures 4.2a and 4.2b are two state machines as generated submodels of traditional and network filesystems in Figures 2.1b and 2.1a with opened as their initial states and ignoring read operation. Both have the same token set $\mathcal{T}(\pi) = \{wa\}$ and $\mathcal{T}(\pi') = \{wa\}$. When

we embed wa of π to wa of π', we mean that the capabilities provided by wa of π should not be distinguishable from the ones provided by wa of π'.

Embedding of States Likewise, regarding bisimilarity of states, we consider opened and to-flush of π on the left (Figure 4.2a) bisimilar to the same state (opened) of the state machine on the right (Figure 4.2b). Although we discuss in-depth in Chapter 5, intuitively speaking, we consider the states to-flush and open of the state machine π that are embedded into the opened state π' such that these states are indistinguishable w.r.t behaviors a client can observe.

4.2 Simulations

After giving the definitions in the previous section, we introduce and explain the parts of our simulation relation. In doing so, we use a graphical representation. We represent the token set and state spaces with ovals separated by labels indicating the clusters of loosely positioned elements of the type on the label per state machine. Legend 3, shown in Figure 4.1, has its labels $\mathcal{T}(\pi)$ and $\mathcal{T}(\pi')$ for the state machines π and π' respectively. Legend 1 represents embedding the state s of the state machine π into the state s' of the state machine π'. Likewise, Legend-2 represented the lifting of token embedding ($\epsilon_{\mathcal{T}}$) to relate the token set T of the state machine π to the token set of T' of the state machine π' which are token sets a client must provide to take a particular transition transition in the state machines π and π' respectively. Legend 4 includes a full circle that represents an element whose value is given in the instance of morphism, a hexagon which represents a placeholder for an existentially quantified element, and an octagon which represents a placeholder for a universally quantified element. To keep the size of Legend in Figure 4.1 reasonable, we show all the embeddings inside clusters with known value placeholders. In Legend 5, we see hexagonal and octagonal nodes with the label n, a natural number greater than 0 denoting the order of quantification of elements: the quantified nodes (filled octagon and hexagon) shown in Legend 4 have the order number 0 – variables in the logical formula that are represented with these elements quantified under the outermost (or first) quantifier in the formula. Legend-6 represents the association of the token set T at the state s in

the state machine π two of which constitute the transition assertion either taken as local (guarantee) or globally (rely) step as shown in Legends 7 and 8 respectively.

Definition 5 (Rely-Surfaces) *The area between* T, s, *and* $\mathsf{s'}$ *in Legend 8 is called a rely-surface. It depicts* $\overset{\mathsf{rely}^*}{\sqsubseteq}_\pi$ *which is a reflexive-transitive closure over a single (inductive) rely (*$\overset{\mathsf{rely}}{\sqsubseteq}_\pi$*) step, whose constructor is*

$$\forall_{T_1,T_2}.\ T_1\ \#\#\ \mathcal{T}(\mathsf{s}) \cup T \rightarrow (\mathsf{s};T_1) \overset{\mathsf{guar}}{\sqsubseteq}_\pi (\mathsf{s'};\mathsf{T}_2) \rightarrow (\mathsf{s};\mathsf{T}) \overset{\mathsf{rely}}{\sqsubseteq}_\pi (\mathsf{s'};\mathsf{T})$$

. The first premise is what we call HGlobalBoundryDom *which asserts the disjointness of interfering tokens (*T_1 *and* T_2*) at a certain state from the clients' capabilities (*T*) and the capabilities hold by the state transition system (*$\mathcal{T}(s)$*). The second premise*

$$(\mathsf{s};T_1) \overset{\mathsf{guar}}{\sqsubseteq}_\pi (\mathsf{s'};\mathsf{T}_2)$$

asserts the possible interfering transition which would not violate the rely step taken between s *and* $\mathsf{s'}$ *against the client with* T

$$(\mathsf{s};\mathsf{T}) \overset{\mathsf{rely}}{\sqsubseteq}_\pi (\mathsf{s'};\mathsf{T})$$

The reflexive transitive closure form of rely-surfaces is shown in Legend 14: a complete rely-surface between x and z can be shown with the knowledge for any rely-reachable y from x, and the *rest of the surface* from y to z which is required for transitive reflexive x-z surface. Regarding the proof machinery, induction on a rely surface can be considered as induction on the reflexive transitive closure of

$$\forall_{T_1,T_2}.\ T_1\ \#\#\ \mathcal{T}(\mathsf{s}) \cup T \rightarrow (\mathsf{s};T_1) \overset{\mathsf{guar}}{\sqsubseteq}_\pi (\mathsf{s'};\mathsf{T}_2) \rightarrow (\mathsf{s};\mathsf{T}) \overset{\mathsf{rely}}{\sqsubseteq}_\pi (\mathsf{s'};\mathsf{T})$$

which would gives us, for $\mathsf{x} = \mathsf{s}$,

$$\forall_{T_1,T_2}.\ T_1\ \#\#\ \mathcal{T}(\mathsf{x}) \cup T \rightarrow (\mathsf{x}; T_1) \overset{\mathsf{guar}}{\sqsubseteq}_{\pi} (\mathsf{y}; \mathsf{T}_2) \rightarrow (\mathsf{x}; \mathsf{T}) \overset{\mathsf{rely}}{\sqsubseteq}_{\pi} (\mathsf{y}; \mathsf{T})$$

together with

$$(\mathsf{y}; T) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (\mathsf{z}; \mathsf{T})$$

For the base case of this induction, we can think of all three lines distanced by a rely step $\overset{\mathsf{rely}}{\sqsubseteq}_{\pi}$ and $\overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi}$ reduced into a single line, which is depicted from top T to the state x — i.e. s' and x are s as shown in Legend 17 4.1.

Legend 13 and Legend 10 represent the disjointness of the token sets T_1 and T_2 for the different state machines and the same state machine, respectively. Finally, Legend 9 represents the state interpretation ($\mathsf{Inv}(\phi)$) of the state s via the interpretation ϕ, i.e., the invariant holding at the state s.

Definition 6 (Bisimulation Zone) *A* bisimulation zone *defines which of the states are considered to be part of the bisimulation. Speaking more concretely, we do not want* closed *(shown in Figure 4.2) to be considered as a part of our bisimulation reasoning. That means, with a given initial state* opened *and initial client token set* $\mathsf{T} = \emptyset$*, we could obtain transitions to the state* closed*, however, we do not consider these transitions as part of the bisimulation, therefore, we want them to be to pruned away as shown in Figures 4.3b and Figure 4.3a.*

Definition 7 (Irrelevancy in Bisimulation) *Since states like* closed *are outside our bisimulation zone, they exhibit* irrelavancy *to our bisimulation relation. Therefore, we need to be able to introduce this irrelevancy in the bisimulation proof. To do so, we rely on the existence of a token such as* cls *and* wa *that is not an element of the client tokens used in the bisimulation zone (e.g.* $\emptyset$ *for our naive example) — please note that* wa *is owned by the state machine not used in client transitions in bisimulation zone. The goal is to distinguish transitions that involve a state outside the bisimulation zone, e.g.* closed*.*

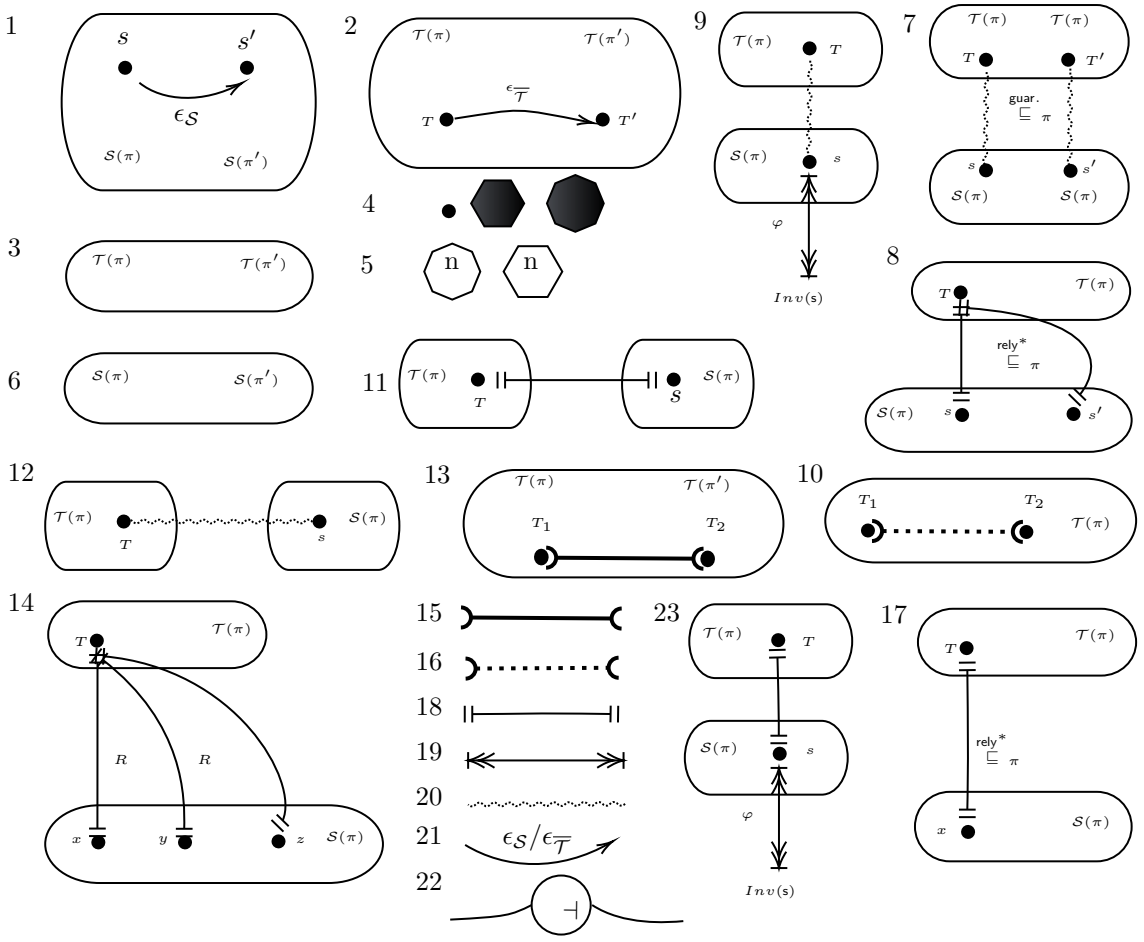
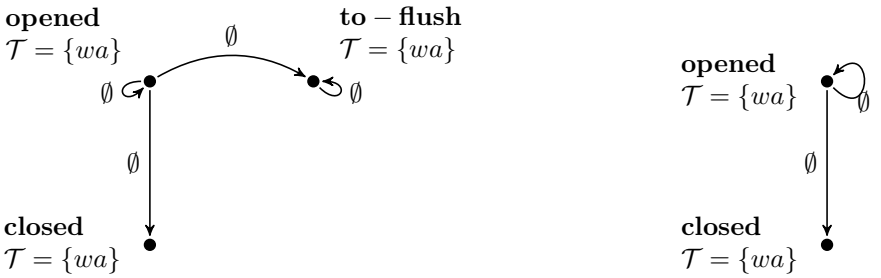


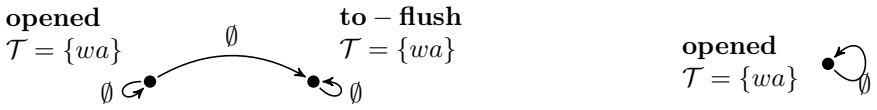
Figure 4.1 Legend for Bisimulation Graphs



(a) Submodel (π) of the distributed I/O protocol generated by **to − flush**

(b) Submodel (π) of traditional I/O protocol generated by **opened**

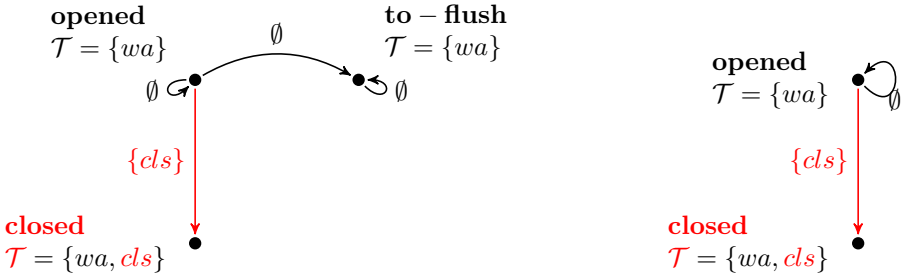
Figure 4.2 Submodels of traditional and distributed file I/O protocols with write accessibility relations.



(a) Submodel (π) of the distributed I/O protocol generated by **to − flush**

(b) Submodel (π) of traditional I/O protocol generated by **opened**

Figure 4.3 Submodels of the traditional and distributed file I/O protocols with write accessibility relations.



(a) Submodel (π) of the distributed I/O protocol generated by **to − flush**

(b) Submodel (π) of traditional I/O protocol generated by **opened**

Figure 4.4 Introducing Irrelavancy .

Tokens like cls are useful because if the proof using the bisimulation owns those tokens, then transitions using that token can effectively be *set aside* soundly and not be used by the bisimulation relation.

4.3 Guarantee in the Bisimulation

One crucial aspect of our simulation relation is *matching guarantee (local) steps* (Theorem *Guarantee Bisim*). This deals with the situation where intuitively, we want to make sure that any step – the one between q' and q" in Figure 4.5 –taken in the target state machine π' has a matching step – the one between s' and s" in Figure 4.5 – in the source state machine π. This match-up is meaningful because if you want to use target-verified code in source-specified settings, it needs to be the case that everything that target code does is allowed by the source spec.

Theorem 4.3.1 (Guarantee Bisim without Invariants)

$$\forall_{q',q,T'}.\, \epsilon_{\overline{\mathcal{T}}}(T,T') \rightarrow \epsilon_{\mathcal{S}}(s,q) \rightarrow (q;T') \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (q';T') \rightarrow$$
$$\forall_{q'',T''}.\, (q';T') \overset{\mathsf{guar.}}{\sqsubseteq}_{\pi'} (q'';T'') \rightarrow$$
$$\exists_{s',s'',T_0',T_0''}.\, (s';T_0') \overset{\mathsf{guar.}}{\sqsubseteq}_{\pi} (s'';T_0'') \wedge$$
$$\epsilon_{\mathcal{S}}(s') = q' \wedge \epsilon_{\mathcal{S}}(s'') = q'' \wedge \epsilon_{\overline{\mathcal{T}}}(T_0',T') \wedge \epsilon_{\overline{\mathcal{T}}}(T_0'',T'')$$

Proof: The proof of Theorem Guarantee Bisim without Invariants 4.3.1, as specified without the association of *invariants* explained in Chapter 5, requires matching the guarantee steps in the target machine (from q' to q") to the guarantee steps in the source (from s' to s").

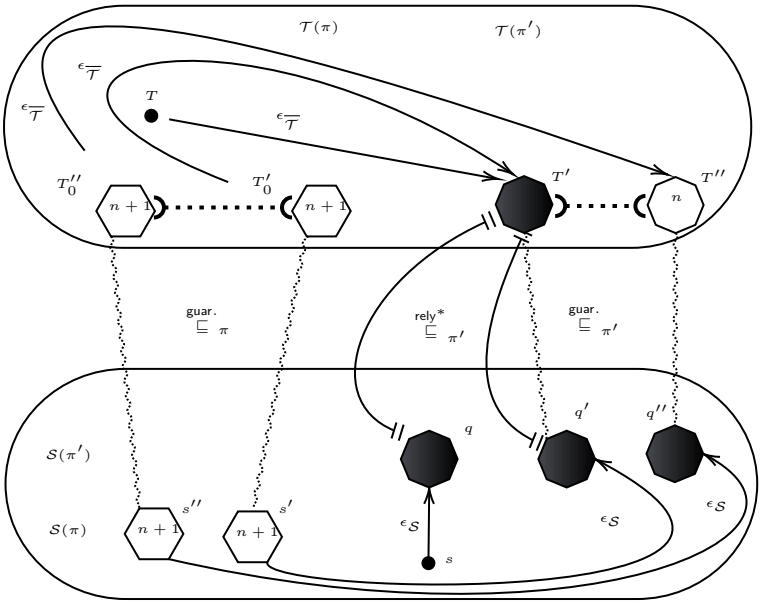


Figure 4.5 Guarantee Bisim without Invariants

The soundness principle of matching guarantee steps relies on *identifiying* q' and *bisimilarity of the matched states*.

Identifying q' Identifying q' of the target state machine π' (shown in Figure 4.5) is the essential piece of reasoning in the guarantee-step matching. This step is valid against any interfering client

action taken from the fixed starting state (s) under the embedded frame token set (T') of the client token set (T) which is given as an input to the simulation relation. The rely-surface $\overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'}$ in Figure 4.5 under the token set T' ensures we only consider q's which are rely-reachable from q against the interference from the client with the token set T'.

Identifying the Domain of the Source Local State To Be Matched In addition to the fact that q' in Figure 4.5 resides in the domain of the rely-reachable states against client actions that can be taken with T' in Figure 4.5,

- the having T' disjoint from from the capabilities in the destination state, T'', (represented as dotted-line in Figure 4.5)
- the reflection of this disjointness to the embedding capabilities in the source state machine as dijoint T'_0 and T''_0
- and, having both disjointness guarantee steps in the previous bullets against an interference with T_1 and T_2

determine the valid domain in which the state q' can be.

$$\mathsf{HGlobalBoundryDom} : \forall_{\mathsf{T}_1,\mathsf{T}_2}.\, \mathsf{T}_1 \#\# \mathcal{T}(q) \cup \mathsf{T}'$$

Bisimilar Matched States The goal asks us not only to prove the existence of the guarantee step in the source state machine ($\overset{\mathsf{guar.}}{\sqsubseteq}_{\pi}$ in Figure 4.5) but also to discharge the fact that the states from/to guarantee steps are taken must be bisimilar: s' and s" in Figure 4.5 are bisimilar to q' and q" respectively via $\epsilon_{\mathcal{S}}$

$$\epsilon_{\mathcal{S}}(s') = q' \text{ and } \epsilon_{\mathcal{S}}(s'') = q''$$

.

Fixed Initial Client Tokens & Start State Our bisimulation relation is fixed on the start state (opened of π in Figure 4.3) from which we consider the sub-model and the client tokens ($\mathsf{T} = \emptyset$. This

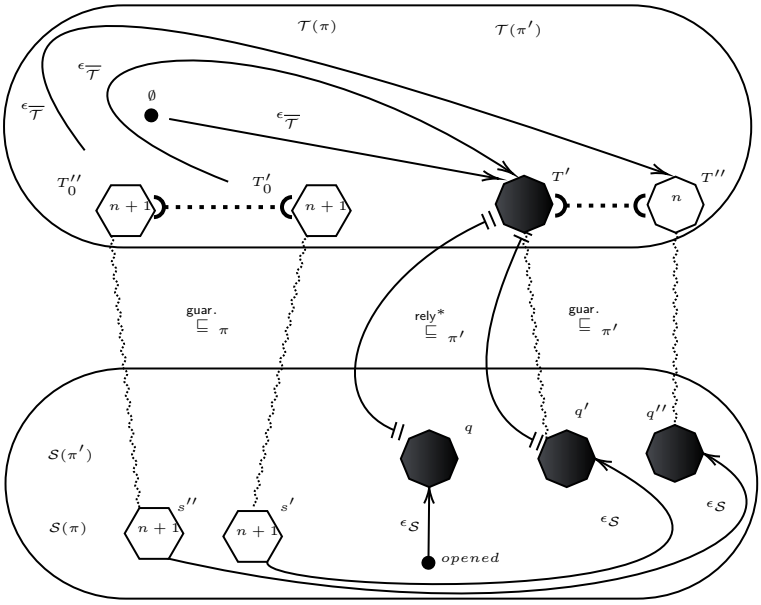


Figure 4.6 Guarantee Bisim (without Invariants) with a Fixed Starting State and a Client Token Set

turns our graph in Figure 4.5 into Figure 4.6 by substituting s of π with opened and T with $\emptyset$. From the embedding state $\epsilon_{\mathcal{S}}(\mathsf{s}, \mathsf{q})$ and the fact that opened is the only state in π', q is also replaced by opened.

After the introduction of the variables and hypothesis, we have the proof context

$$\mathsf{HBaseTkembed} : \epsilon_{\overline{\mathcal{T}}}(\emptyset, T')$$

$$\mathsf{HStEmbed} : \epsilon_{\mathcal{S}}(\mathsf{opened}) = \mathsf{opened}$$

$$\mathsf{Hframesteps} : (\mathsf{opened}; T') \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (q'; T')$$

$$\mathsf{Hsteptarget} : (q'; T') \overset{\mathsf{guar}.}{\sqsubseteq}_{\pi'} (q''; T'')$$

and the goal

$$\begin{aligned} & \mathsf{Goalmatchstep} : (s'; T_0') \overset{\mathsf{guar}.}{\sqsubseteq}_{\pi} (s''; T_0'') \\ & \wedge \mathsf{GoalStembedStart} : \epsilon_{\mathcal{S}}(s') = q' \\ \exists_{s', s'', T_0', T_0''}. \; & \wedge \mathsf{GoalStembedEnd} : \epsilon_{\mathcal{S}}(s'') = q'' \\ & \wedge \mathsf{GoalTkEmbedStart} : \epsilon_{\overline{\mathcal{T}}}(T_0', T') \\ & \wedge \mathsf{GoalTkEmbedEnd} : \epsilon_{\overline{\mathcal{T}}}(T_0'', T'') \end{aligned}$$

Assuming token embedding ($\epsilon_{\overline{\mathcal{T}}}$) as set equivalence ($\equiv$), the core piece of our proof is *validity* of q'. We know that it is reachable from the start state, i.e., Hframesteps, against the client interference determined by the chosen initial client token set, HBaseTkembed. However, we need to obtain more information for the valid q' by analyzing the rely surface in Figure 4.6 with inversion on Hframesteps (with initial client token set T' as $\emptyset$)

$$\begin{aligned} \forall_{T_1, T_2}. \; T_1 \; \#\# \, \mathcal{T}(\mathsf{opened}) \cup \emptyset \rightarrow \\ (\mathsf{opened}; T_1) \overset{\mathsf{guar}}{\sqsubseteq}_{\pi} (q'; T_2) \rightarrow \\ (\mathsf{opened}; \emptyset) \overset{\mathsf{rely}}{\sqsubseteq}_{\pi} (q'; \emptyset) \end{aligned}$$

that extends the proof context with:

$$\begin{aligned} & \mathsf{Hdisjframetok} : \; T_1 \; \#\# \, \mathcal{T}(\mathsf{opened}) \cup \emptyset \\ & \mathsf{Hrelysoundness} : ((\mathsf{opened}; T_1) \overset{\mathsf{guar}}{\sqsubseteq}_{\pi} (q'; T_2) \\ & \mathsf{Hframestep} : (\mathsf{opened}; \emptyset) \overset{\mathsf{rely}}{\sqsubseteq}_{\pi} (q'; \emptyset) \end{aligned}$$

Rely-Surface Intuition on the Proof Task *Concretely* – Induction on $\overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'}$ As discussed, the crux point of proof goal is to know what q' is. This requires analyzing the rely-surface between q and q' shown in Figure 4.7 — i.e., Hframestep with T' as an empty set. The surface analysis is performed on single-relying-step reachable valid states y from x which is q which can only be opened based on the design of π' and HStEmbed. The identification of q', which is known to be equal to z by inversion on Hframesteps, should be reached from these single rely reachable y states by taking

other valid rely-steps (hypothesis Hframesteps)

$$\mathsf{Hframestepleft} : (\mathsf{y}; \emptyset) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (q'; T') \qquad where \qquad \mathsf{q'} = \mathsf{z}$$

Intuitively speaking, the identification of q' can be thought of as capturing the rely-surface (Hframesteps) depicted in Figure 4.6, by single-rely-reachability (Hframestep) and the rely-surfrace left (Hframestepsleft) depicted in Figure 4.7.

Base-Case is the reflexive case of our rely surface. This case can be the case in which the rely step has both y and z to be opened — concretely speaking, there is no rely step taken to y on the rely surface: there exists no Hframestep in the proof context. Because it is outside the bisimulation zone due to being unreachable without non-withheld tokens, per Definition 6), the possible s", either to-flush or opened of π as the embedding states of opened of π', requires us to pick T_0'' as an empty set, since at both of these possibilities the client, by the design of π, can have the empty token set. Knowing that the initial client token set T is empty, and consequently having the hypothesis HBaseTkembed in the proof context, we also pick T_0' as an empty set. Picking opened for both s' and s'', the proof goal becomes

$$\begin{aligned}
&\mathsf{Goalmatchstep} : (\mathsf{opened}; \emptyset) \overset{\mathsf{guar.}}{\sqsubseteq}_{\pi} (\mathsf{opened}; \emptyset) \\
&\wedge \mathsf{GoalStembedStart} : \epsilon_{\mathcal{S}}(\mathsf{opened}) = \mathsf{opened} \\
&\wedge \mathsf{GoalStembedEnd} : \epsilon_{\mathcal{S}}(\mathsf{opened}) = q'' \\
&\wedge \mathsf{GoalTkEmbedStart} : \epsilon_{\overline{\mathcal{T}}}(\emptyset, \emptyset) \\
&\wedge \mathsf{GoalTkEmbedEnd} : \epsilon_{\overline{\mathcal{T}}}(\emptyset, T'')
\end{aligned}$$

Overall Proof for Base-Case Goalmatchstep requires proving

$$\mathsf{Hdisj} : \emptyset \# \# \mathcal{T}(opened)$$

$$\mathsf{Hnextdisj} : \emptyset \# \# \mathcal{T}(opened)$$

$$\mathsf{Hprim} : \mathsf{opened} \rightsquigarrow \mathsf{opened}$$

$$\mathsf{Hlocalpreserve} : \mathcal{T}(opened) \cup \emptyset \equiv \mathcal{T}(opened) \cup \emptyset$$

where we know

$$\mathcal{T}(opened) = \{wa\}$$

in Figures 4.3a and 4.3a, and Goalmatchstep follows from GoalStembedStart and $\epsilon_{\mathcal{S}}$. GoalTkEmbedStart follows from HBaseTkembed.

Inductive-Case Based on the intuition we built for the rely-surfaces relevant to this proof, the inductive case is considered to be the construction of the rely surface in Figure 4.7 with the immediately reachable state y to reach to the final state q'= z such that

$$\mathsf{IFrameSurface} : (\mathsf{opened}; \emptyset) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (z; \emptyset)$$

$$\mathsf{IHBase} : \mathsf{x} = \mathsf{opened}$$

$$\mathsf{IHBaseTkembed} : \epsilon_{\overline{\mathcal{T}}}(\emptyset, \emptyset)$$

$$\mathsf{IHStEmbed} : \epsilon_{\mathcal{S}}(\mathsf{opened}) = \mathsf{opened}$$

$$\mathsf{IHframestep} : (\mathsf{opened}; \emptyset) \overset{\mathsf{rely}}{\sqsubseteq}_{\pi'} (y; \emptyset)$$

$$\mathsf{IHframesteps} : (y; \emptyset) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (z; \emptyset)$$

$$\mathsf{IHsteptarget} : (z; \emptyset) \overset{\mathsf{guar.}}{\sqsubseteq}_{\pi'} (q''; T'')$$

in addition to what we know from IHframestep

$$\mathsf{IHdisj} : \mathsf{T}_1 \#\# \mathcal{T}(\mathit{opened})$$

$$\mathsf{IHnextdisj} : \mathsf{T}_2 \#\# \mathcal{T}(y)$$

$$\mathsf{IHprim} : \mathsf{opened} \rightsquigarrow \mathsf{y}$$

$$\mathsf{IHlocalpreserve} : \mathcal{T}(\mathit{opened}) \cup \mathsf{T}_1 \equiv \mathcal{T}(y) \cup \mathsf{T}_2$$

$$\mathsf{IHstep} : (\mathsf{opened}; \mathsf{T}_1) \overset{\mathsf{guar}}{\sqsubseteq}_{\pi'} (y; \mathsf{T}_2)$$

- **Inductive-Case-Open** (Figure 4.7): obtaining z as opened, and the proof goal

$$\mathsf{Goalmatchstep} : (\mathsf{opened}; \emptyset) \overset{\mathsf{guar.}}{\sqsubseteq}_{\pi} (\mathsf{opened}; \emptyset)$$

$$\wedge \mathsf{GoalStembedStart} : \epsilon_{\mathcal{S}}(\mathsf{opened}) = \mathsf{opened}$$

$$\wedge \mathsf{GoalStembedEnd} : \epsilon_{\mathcal{S}}(\mathsf{opened}, q'')$$

$$\wedge \mathsf{GoalTkEmbedStart} : \epsilon_{\overline{\mathcal{T}}}(\emptyset, \emptyset)$$

$$\wedge \mathsf{GoalTkEmbedEnd} : \epsilon_{\overline{\mathcal{T}}}(\emptyset, T'')$$

  includes the q'' on which we induce, end up having two cases: 1. q'' is opened and the goal is trivial and discussed 2. q'' is closed, we need to create a contradiction by relying on the following facts

  - Any client taking a step into a state outside the bisimulation zone the bisimulation (see Definition 6) has to utilize a set of token, e.g. $\{cls\}$, that is disjoint to the set of all tokens that can be used by a client inside the bisimulation zone zone, $T' = \emptyset$ of π' and T of π
  - Although the step taken from opened to closed — IHsteptarget — requires the client token set for to be $\{cls\}$, however, out Guarantee Bisim relation enforces it to be embedding of the initial token set, $T' = \emptyset$. This creates a contradiction (see Definition 7).

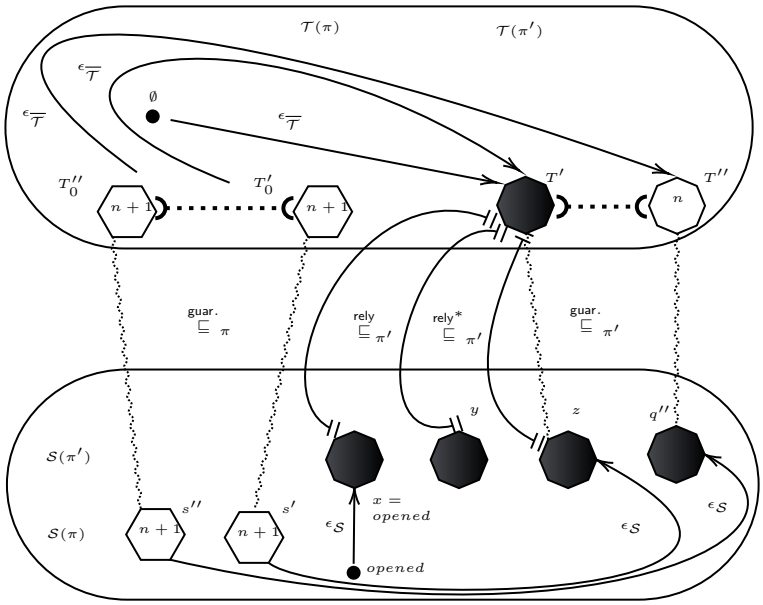


Figure 4.7 Induction on the Rely-Steps of Guarding Condition of Guarantee Bisim (without Invariant)

- **Inductive-Case-Closed** (Figure 4.7): obtaining z as closed

$$\mathsf{Goalmatchstep} : (\mathsf{opened}; \emptyset) \overset{\mathsf{guar.}}{\sqsubseteq}_{\pi} (\mathsf{closed}; \emptyset)$$

$$\wedge \mathsf{GoalStembedStart} : \epsilon_{\mathcal{S}}(\mathsf{opened}) = \mathsf{closed}$$

$$\wedge \mathsf{GoalStembedEnd} : \epsilon_{\mathcal{S}}(\mathsf{closed}) = q''$$

$$\wedge \mathsf{GoalTkEmbedStart} : \epsilon_{\overline{\mathcal{T}}}(\emptyset, \emptyset)$$

$$\wedge \mathsf{GoalTkEmbedEnd} : \epsilon_{\overline{\mathcal{T}}}(\emptyset, T'')$$

would be invalid by IFrameSurface and IHsteptarget.

□

4.4 Rely in the Bisimulation

Bisimulation relation must not drop any valid interference, and match them on the other state machine to make sure that start-end pairs must be preserved exactly. As a result, the bisimulation relation ensures that client specifications (both on the source and target specifications) refer to the

valid abstract states on both target and source STSes. Considering the direction from source to target STS, for every possible state in the source rely, the postcondition of rely-moves to that state must imply the rely-moves of the corresponding transitions in the target state state machine. It is essential for file protocol example as we would like to have

$$\varphi_{\mathsf{distributedfile}}(\mathsf{to-flush}) \vdash \varphi_{\mathsf{opened}}(\mathsf{opened})$$

preserved by the bisimulation.

Theorem Rely Bisim 4.4.1 asserts the validation of any rely steps taken in one state machine from the perspective of the other state machine. More concretely speaking, the rely steps constructing the rely surface between any s and s' in Figure 4.8 must be matched by the rely surface between any embedded state of s and s', shown as s'_1 and s'_1, respectively, in Figure 4.8. These rely surfaces are constructed against client interference determined by the token set defining capabilities of the client. For example T (given as a value to the bisimulation relation instance) and T_1 represent the capabilities of the client at the states s and s_1 in the state machines π and π', respectively.

Theorem 4.4.1 (Rely Bisim)

$$\forall s'.(s;T) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (s';T) \leftrightarrow (\forall_{s_1,s'_1,T_1}.\, \epsilon_{\mathcal{S}}(s,s_1) \to \epsilon_{\mathcal{S}}(s',s'_1) \to \epsilon_{\overline{\mathcal{T}}}(T,T_1) \to (s_1;T_1) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (s'_1;T_1)$$

Proof: First, we start by proving the implication from left to right.

$$\forall s'.(s;T) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (s';T) \to (\forall_{s_1,s'_1,T_1}.\, \epsilon_{\mathcal{S}}(s) = s_1 \to \epsilon_{\mathcal{S}}(s') = s'_1 \to \epsilon_{\overline{\mathcal{T}}}(T,T_1) \to (s_1;T_1) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (s'_1;T_1)$$

This direction requires us to show that knowing the validity of rely-steps taken from s reaching to s' against a client interference with tokens T, which is, as we mentioned in the previous chapter, the client token set for the initial state. This frame defines the rely surface ($\overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi}$) on the left matching

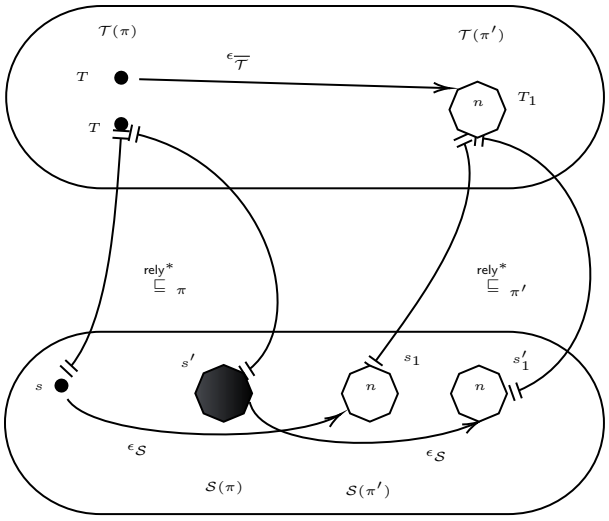


Figure 4.8 Theorem Rely Bisim

the rely steps on the right — the steps constructing the surface $\overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'}$ on the right.

As we know, our initial starting state, s of π, is open, and initial client token set, T, is an empty set. With these values, our implication (shown in Figure 4.9)

$$\forall s'.(\mathsf{opened};\emptyset) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (s';\emptyset) \to (\forall_{s_1,s'_1,T_1}.\, \epsilon_{\mathcal{S}}(\mathsf{opened} = s_1 \to \epsilon_{\mathcal{S}}(s') = s'_1 \to \epsilon_{\overline{\mathcal{T}}}(\emptyset, T_1) \to (s_1;T_1) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (s'_1;T_1)$$

and by introducing the rely steps, we end up having the proof context with hypothesis

$$\mathsf{IHrtc} : (\mathsf{opened};\emptyset) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (s';\emptyset)$$

.

As we know, $\overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi}$ is a reflexive-transitive closure of a single (inductive) rely ($\overset{\mathsf{rely}}{\sqsubseteq}_{\pi}$) step

$$\begin{array}{l} \forall_{\mathsf{T}_1,\mathsf{T}_2}.\ \mathsf{T}_1\ \#\#\ \mathcal{T}(\mathsf{opened}) \cup \mathsf{T} \to \\ \qquad (\mathsf{opened};T_1) \overset{\mathsf{guar}}{\sqsubseteq}_{\pi} (s';T_2) \to \\ \qquad\qquad (\mathsf{opened};\mathsf{T}) \overset{\mathsf{rely}}{\sqsubseteq}_{\pi} (s';\mathsf{T}) \end{array}$$

that can be taken against the client capabilities represented by the token set $\mathsf{T} = \emptyset$ without

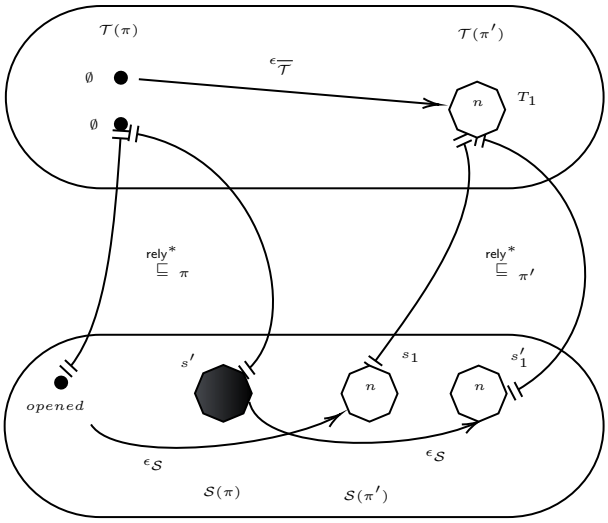


Figure 4.9 Fixed Frame Tokens and Initial State in Rely Bisim

violating any guarantee step taken by the client from the state opened with any capability — HGlobalBoundryDom in Theorem Rely Bisim.

$$\mathsf{HGlobalBoundryDom} : \mathsf{T}_1 \ \#\# \ \mathcal{T}(\mathsf{opened}) \cup \mathsf{T} \quad \text{where} \quad \mathsf{T} = \emptyset$$

Again, we use induction to compute the rely-surface against the client with capabilities T. In Figure 4.10, we see the hypothesis Hframestep such that for any single rely-reachable valid state, y in Figure 4.10, from which we know that we can take other valid rely-steps to reach to the state, z, that is, the source state of the guarantee step to be matched. With regard to the proof machinery, an induction on IHrtc would change the proof state as shown in Figure 4.10: based on the knowledge of a single rely-step taken from the initial state of the simulation, opened of π, to a state (y) from which we know that taking the required rely-steps (the rely-surface between y and z) enables reaching from y to z. In other words, the matching rely surface in the target state machine π' is checked against any rely step that builds the rely surface in the state machine π.

Base-Case be thought of as the case in which the rely step has both y and z as opened — intuitively speaking, there is no iteration of a rely surface through a rely step taken to y. In the proof machinery, a single reflexive transitive step (rtc_once) in the target state machine π' at opened state with the

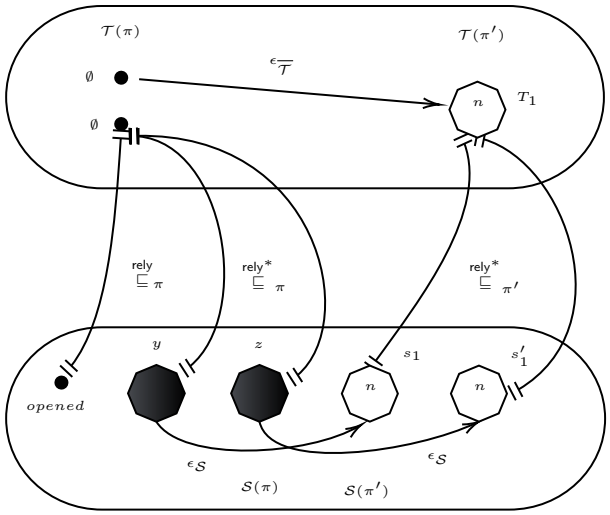


Figure 4.10 Induction on Rely Bisim

knowledge $T = \emptyset$ and $\mathcal{T}(\mathsf{opened}) = \emptyset$ leaves the goal,

$$\begin{array}{ll} \mathsf{HBaseStembed0}: & \epsilon_{\mathcal{S}}(\mathsf{opened}) = s_1) \\ \mathsf{HBaseStembed1}: & \epsilon_{\mathcal{S}}(\mathsf{opened}) = s_1') \\ \mathsf{HTkembed}: & \epsilon_{\overline{\mathcal{T}}}(\emptyset, T_1) \\ \mathsf{GoalBase}: & (s_1; T_1) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (s_1'; T_1) \end{array}$$

and applying the constructor of the inductive rely-step relation (Frame_step) with the universally quantified tokens for the guarantee step as $T_1 = \emptyset$ $T_2 = \emptyset$, we end up having the premises of the rely-step relation in the proof goal

$$\begin{array}{ll} \mathsf{Goal1}: & \emptyset \# \# \mathcal{T}(opened) \cup \emptyset \\ \mathsf{Goal2}: & (\mathsf{opened}; \emptyset) \overset{\mathsf{guar}}{\sqsubseteq}_{\pi} (\mathsf{opened}; \emptyset) \end{array}$$

Inductive-Case

$$\begin{array}{ll} \mathsf{Hframestep}: & (\mathsf{opened}; \emptyset) \overset{\mathsf{rely}}{\sqsubseteq}_{\pi} (y; \emptyset) \\ \mathsf{Hframesteps}: & (y; \emptyset) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (z; \emptyset) \\ \mathsf{IIHrtc}: & \epsilon_{\mathcal{S}}(y) = s_1 \rightarrow \epsilon_{\mathcal{S}}(z) = s_1' \wedge \epsilon_{\mathcal{T}}(\emptyset, T_1) \wedge (s_1; T_1) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (s_1'; T_1) \end{array}$$

and the goal

$$\mathsf{GoalInductive}: \quad \epsilon_{\mathcal{S}}(z) = s_1' \wedge \epsilon_{\mathcal{T}}(\emptyset, T_1) \wedge (s_1; T_1) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (s_1'; T_1)$$

As expected we would apply IIHrtc which leaves us with the following proof goal

$$\mathsf{Goalrtc1}: \quad \epsilon_{\mathcal{S}}(y) = s_1$$

where we know that $\epsilon_{\mathcal{S}}(s_1) = x$.

To do so, we need more information on the states. Inversion on Hframestep (then from the guarantee step inside the rely-step relation) gives us

Hstep:	$(\mathsf{opened}, T_1) \overset{\mathsf{guar.}}{\sqsubseteq}_{\pi} (y, T_2)$
Hdisj:	$T_1 \#\# \mathcal{T}(\mathsf{opened}) \cup \emptyset$
Hlocaldisj:	$\mathcal{T}(\mathsf{opened}) \#\# T_1$
Hprim:	$\mathsf{opened} \rightsquigarrow y$
Hnextdisj:	$\mathcal{T}(y) \#\# T_2$
Hlocalpreserve:	$\mathcal{T}(\mathsf{opened}) \cup T_1 \equiv \mathcal{T}(y) \cup T_2$

we need to do inversion on the primitive step relation assumption, Hprim, which gives us a single step taken from opened to either opened and to-flush of π, and leaves us to prove Goalrtc1 for both $y = \mathsf{opened}$ and $y = \mathsf{to\text{-}flush}$ as shown in Figures 4.11a and 4.11b, respectively. In addition to the cases shown in Figures 4.11a and 4.11b, we would also have the proof context for $\mathsf{y} = \mathsf{closed}$ with the frame step

$$\mathsf{Hframestep}: \quad (\mathsf{opened}; \emptyset) \overset{\mathsf{rely}}{\sqsubseteq}_{\pi} (y; \emptyset) \qquad \mathsf{where} \quad \mathsf{y} = \mathsf{closed}$$

such that after applying IIHrtc to the inductive goal GoalInductive we end up having each of the

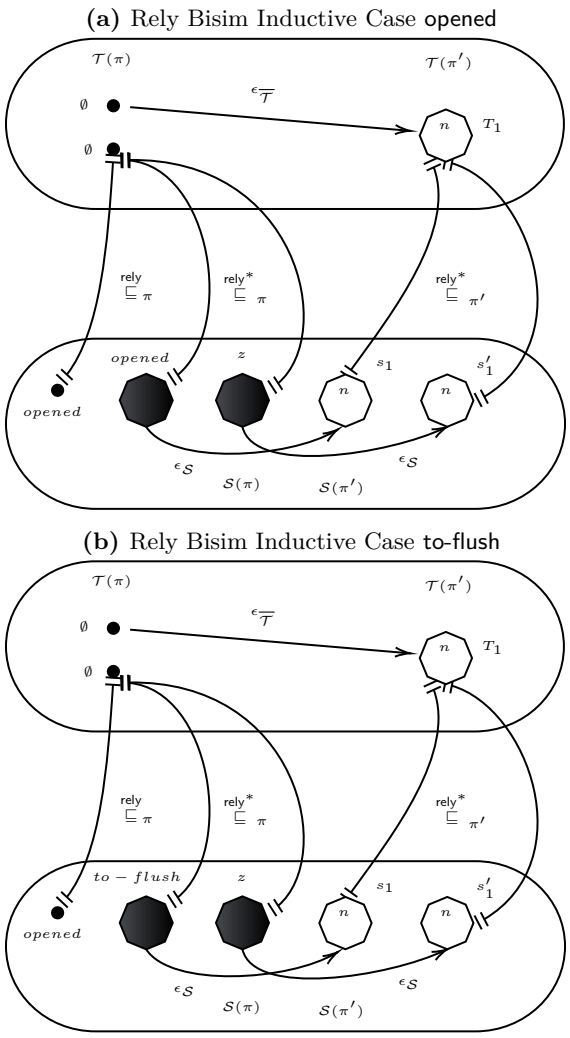


Figure 4.11 Rely Bisim Inductive Cases

following proof obligations

$$\mathsf{GoalInductiveO}: \quad \exists_{s_1'}.\, \epsilon_{\mathcal{S}}(\mathsf{opened} = s_1' \wedge \epsilon_{\mathcal{T}}(\emptyset, T_1) \wedge (\mathsf{opened}; T_1) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (s_1'; T_1)$$

$$\mathsf{GoalInductiveF}: \quad \exists_{s_1'}.\, \epsilon_{\mathcal{S}}(\mathsf{to-flush}) = s_1' \wedge \epsilon_{\mathcal{T}}(\emptyset, T_1) \wedge (\mathsf{opened}; T_1) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (s_1'; T_1)$$

$$\mathsf{GoalInductiveC}: \quad \exists_{s_1'}.\, \epsilon_{\mathcal{S}}(\mathsf{closed}) = s_1' \wedge \epsilon_{\mathcal{T}}(\emptyset, T_1) \wedge (\mathsf{opened}; T_1) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (s_1'; T_1)$$

These goals are trivial based on the following points:

- $\epsilon_{\mathcal{S}}$ embeds closed of π to closed of π'
- and there is a client step from opened to closed so that we can apply framing (i.e. Frame_step)

in the goal with interferring token $\{cls\}$ at source state opened and the $\emptyset$ at the destination opened as shown in Figure 4.4b.

- For irrelevant cases in the goal such as $\epsilon_{\mathcal{S}}(\mathsf{closed}) = \mathsf{opened}$ and $\epsilon_{\mathcal{S}}(\mathsf{closed}) = \mathsf{to-flush}$, *irrelevancy* can be introduced (defined in Definition 7 Irrelevancy in Bisimulation) via specializing Hlocaldisj and Hdisj with either wa or cls both of which are irrelevant to any client actions in the bisimulation zone.

□

CHAPTER 5

INVARIANTS

In this chapter, we introduce the complete bisimulation relation is shown in Figure 5.1. On top of the intuition we built with the proofs of Guarantee Bisim Theorem 4.3.1 and Rely Bisim Theorem 4.4.1, we introduce Guarantee ($\sim_{\mathsf{Guarantee}}$) and Rely ($\sim_{\mathsf{Rely}}$) laws in our bisimulation relation 5.1 which decorate already discussed reasoning in these proofs with the the concept of *invariant*. The other change in guarantee steps reasoning is that the laws of our bisimulation does not consider single target guarantee step ($\overset{\mathsf{guar}}{\sqsubseteq}_{\pi'}$) to be matched (discussed in Guarantee Bisim Theorem 4.3.1), but instead it considers closure ($\overset{\mathsf{guar}^*}{\sqsubseteq}_{\pi'}$) of target guarantee steps to be bisimulated.

$$
\mathcal{M}(\pi,\pi',\varphi,\varphi',s,\mathsf{T},\mathsf{U}) = \begin{cases}
\epsilon_{\mathcal{S}}: & \mathcal{S}(\pi) \mapsto \mathcal{S}(\pi') \\
\epsilon_{\mathcal{T}}: & \mathcal{T}(\pi) \mapsto \mathcal{T}(\pi') \\
\sim_{\mathsf{Rely}}: & \forall\, s'.\ (s\ \mathsf{T}) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (s';\mathsf{T}) \leftrightarrow \\
 & (\epsilon_{\mathcal{S}}(s);\epsilon_{\overline{\mathcal{T}}}(\mathsf{T})) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (\epsilon_{\mathcal{S}}(s');\epsilon_{\overline{\mathcal{T}}}(\mathsf{T})) \\
\sim_{\mathsf{Guarantee}}: & \forall\, q'\ .\ (\epsilon_{\mathcal{S}}(s);\epsilon_{\overline{\mathcal{T}}}(\mathsf{T})) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (\epsilon_{\mathcal{S}}(q');\epsilon_{\overline{\mathcal{T}}}(\mathsf{T})) \rightarrow \\
 & \qquad \forall\, q'',\ \mathsf{T}''\ .\ (q';\epsilon_{\overline{\mathcal{T}}}(\mathsf{T})) \overset{\mathsf{guar}^*}{\sqsubseteq}_{\pi'} (q'';\mathsf{T}'') \rightarrow \\
 & \qquad \exists\, s'\ s''\ \mathsf{T0}'\ \mathsf{T0}''\ .\ (s';\mathsf{T0}') \overset{\mathsf{guar}^*}{\sqsubseteq}_{\pi} (s'';\mathsf{T0}'') \wedge \\
 & \qquad\qquad \epsilon_{\mathcal{S}}(s') = q' \wedge \epsilon_{\mathcal{S}}(s'') = q'' \wedge \\
 & \qquad\qquad (\epsilon_{\overline{\mathcal{T}}}(\mathsf{T0}')) = (\epsilon_{\overline{\mathcal{T}}}(\mathsf{T})) \wedge \\
 & \qquad\qquad (\epsilon_{\mathcal{T}}(\mathsf{T0}')) = \mathsf{T}'' \wedge \varphi'(q'') \vdash \varphi(s'') \\
\sim_{\mathsf{Tolerance}} & \forall\, s'\ .\ (s\ \mathsf{T}) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (s';\mathsf{T}) \leftrightarrow \varphi(s') \vdash \varphi'(\epsilon_{\mathcal{S}}(s'))
\end{cases}
$$

Figure 5.1 Bisimulation Relation

5.1 Remarks on Invariants and Interacting with STS

Modern separation logics such as CaReSL [156] and Iris [91] provide logical mechanisms for accessing and restoring invariants. To refer to an invariant, we define a function – state interpretation function – that allows us to access the invariant associated with that state. For example, φ_{file} as the state-interpretation function for the state machine π in Figure 4.2a could be given for any file handle ℓ, concrete file state $\mathsf{fs} \in \mathsf{N} * \mathsf{N}$ describing a file's status and identification, and file content $(\mathsf{R} : \mathsf{iProp}\ \Sigma)$

$$\varphi_{\mathsf{distributedfile}}\ (\ \ell\ ,\ \mathsf{R}\ )(\ s\ ) \triangleq \left\{ \begin{array}{ll} \mathsf{match\ s\ with} & \\ \mathsf{to-flush} \Rightarrow & \mathsf{R} * \exists\ \mathsf{fs}.\ \mathsf{isValidDirty}(\mathsf{fs}) * \ell \mapsto (\mathsf{fs}.id, \mathsf{fs.status} = \mathsf{dirty}) \\ \mathsf{opened} \Rightarrow & \mathsf{R} * \exists\ \mathsf{fs}.\ \mathsf{isValid}(\mathsf{fs}) * \ell \mapsto (\mathsf{fs}.id, \mathsf{clean}) \\ \mathsf{closed} \Rightarrow & \exists\ \mathsf{fs}.\ \mathsf{isValidClosed}(\mathsf{fs}) * \ell \mapsto (\mathsf{fs}.id, \mathsf{closed}) \end{array} \right\}$$

$\varphi_{distributedfile}$ associates the abstract state s with a simple cocrete file state (fs) mentioned in an invariant-per-state that represents the file content (e.g., $\mathsf{isValidClean}$) and the fact specific to the file's *status* (e.g.,clean). Because it is connecting the physical state to an abstract one used in the specification, defining the state interpretation is the first essential step to check client code follows the specification encoded as state machines, but then we need to define the constructions and rules defining the protocol and orchestrating accessibility to it.

Representing STS Invariants in Iris We already gave a definition for

$$\boxed{s; T}^{n}_{\pi} \approx \exists s'.\, s \rightsquigarrow^{*} s' \wedge \varphi(s') \wedge \mathsf{RegionOwned}(n, \mathcal{T}(s')) \wedge \mathsf{LocalToks}(T)$$

which asserts that an invariant holds the persistent truth per state ($s \in \mathcal{S}$), and make it accessible via an state interpretation function (φ Legend 19 in Figure 4.1). In this definition from CaReSL [156], token ownership of the state machine and the client are explicitly and separately mentioned in the predicates $\mathsf{RegionOwned}$ and $\mathsf{LocalToks}$, respectively. In Iris [91], we dismantle what $\boxed{s; \mathsf{T}}_{\pi}$

$$\frac{\text{STSAlloc}}{\varphi(s) \Rrightarrow \exists\gamma.\, \boxed{\varphi}^{\gamma}_{\pi} * \boxed{s; \mathsf{AllTokens} \setminus \mathcal{T}(s)}^{\gamma}} \qquad \frac{\text{STSOpen}}{\boxed{\varphi}^{\gamma}_{\pi} * \boxed{s; \mathsf{T}}^{\gamma} \Rrightarrow (\exists\, s'.\, \ulcorner(\mathsf{s0}, \mathsf{T}) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (\mathsf{s}', \mathsf{T})\urcorner * \varphi(s) * \forall \mathsf{sl}',\ \mathsf{T}'.\, \ulcorner(\mathsf{s}', \mathsf{T}) \overset{\mathsf{guar.}^*}{\sqsubseteq}_{\pi} (\mathsf{sl}', \mathsf{T}')\urcorner * \varphi(\mathsf{sl}') \Rrightarrow \boxed{\mathsf{sl}; \mathsf{T}'}^{\gamma})}$$

Figure 5.2 Iris STS Library [91] simplified with later modality and invariant masks omitted

semantically asserts in such a way that the boxed assertion

$$\boxed{\varphi}^{\gamma}_{\pi} \triangleq \exists\, \mathsf{s}.\, \varphi(s) * \mathsf{sts\_auth}(\ \mathsf{s},\ \emptyset,\ \gamma,\ \pi\ )$$

both controls the access to the invariant via holding the *state machine*'s authorative ownership, sts_auth, and the invariant itself accessed for a state s via the state interpretation (φ). The authorative ownership, as shown sts_auth(s, $\emptyset$, γ, π), is always inside the invariant ($\boxed{\varphi}^{\gamma}_{\pi}$, and once it is accessed it does not allow any other permissible steps via any capability by setting the token set to $\emptyset$. As a, the boxed assertion definition we gave early in this part of the thesis, which considers permissible steps by referring to the capabilities represented by T in it, is different than how Iris encodes STS invariants.

Fragmental Tokens for Invariant Access Fragmental ownership of tokens, $\boxed{s; \mathsf{T}}^{\gamma}$ (shown in Legend 9 and 23 for rely and guaratee relations, respectively), leaves the client with capabilities that enable *borrowing* the resources which are accessed in an *atomic* program action and represented with $\varphi(\mathsf{s})$ and $\mathsf{Inv}(\mathsf{s})$ respectively (shown in Legend 9 and 23 for rely and guaratee relations, respectively), and step into a new state (for example, s' in the definition given in Section 2.1) in which we can restore the invariant.

We refer to invariants of a system per state; for example, a file's dirty bit in the $\mathsf{to-flush}$ state is *set*. To do so, we use an interpretation of a state ϕ, which is a function of a state to a proposition representing the invariant holding on the state. Legend 9 in Figure 4.1 shows this association: $\boxed{s; \mathsf{T}}_{\pi}$ contains an invariant (Inv) in state s represented by φ.

Rules for Allocating, Opening and Closing STS Invariants The rules shown in Figure 5.2 allow for the allocation (STSAlloc) and access (STSOpen) of resources represented with state machine invariants. Introducing an invariant holding at the state s – $\varphi(\mathsf{s})$ – can be done by a *view-shift* in the ghost state that puts the invariant and associated authorative ownership under $\boxed{\varphi}^{\gamma}_{\pi}$ and reveals all the capabilities that the state machine does not hold at the state s to the client through fragmental ownership – $\boxed{s; \mathsf{AllTokens} \setminus \mathcal{T}(s)}^{\gamma}$ – under a new name γ. Accessing the invariant inside the state machine, requires the considering client interference, $\boxed{s; \mathsf{T}}^{\gamma}$ to infer the next *sound state* sl at which the invariant can be restored, and into which client can take a step $\boxed{\mathsf{sl}; \mathsf{T}'}^{\gamma}$. As we discuss in the previous chapter, *soundness* is ensured by the rely

$$(\mathsf{s0}, \mathsf{T}) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (\mathsf{s}', \mathsf{T})$$

and the guaratee

$$(\mathsf{s}', \mathsf{T}) \overset{\mathsf{guar.}^*}{\sqsubseteq}_{\pi} (\mathsf{sl}', \mathsf{T}')$$

relations.

5.2 Tolerance of Invariants

The first extension in our bisimulation relation in Figure 5.1 focuses on how we relate invariants of one machine to another against a client interference to the other machine – the law of Tolerance in Figure 5.1. In other words, with the invariant tolerance, we view the interference as occuring in the bisimulation target, the actual resources owned in the bisimilar states (s' and $\epsilon_{\mathcal{S}}(\mathsf{s})'$) justify the same actions, as shown in Figure 5.3.

For every possible state in the original rely, the postcondition of rely-moves to that state must imply the rely-moves of the corresponding transitions in the target state state machine (we already know from above that those transitions exist – the law of Rely in Figure 5.1). To speak concretely, we already gave the state interpretation function, $\varphi_{\mathsf{distributedfile}}$, for the state machine π. An expected

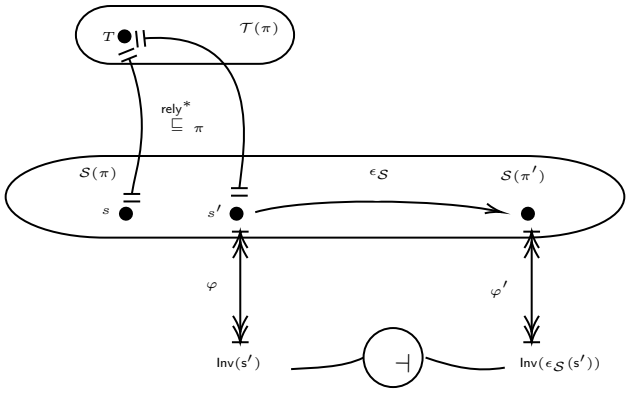


Figure 5.3 Tolerance of Invariants with initial state s as opened

interpretation function of the state machine for the traditional file protocol, π', would be

$$\varphi_{\mathsf{file}}\ \ell\ \mathsf{R}\ s\ \triangleq \left\{ \begin{array}{ll} \mathsf{match\ s\ with} & \\ \mathsf{opened} \Rightarrow & \mathsf{R} * \exists\ \mathsf{fs.\ isValid(fs)} * \ell \mapsto (\mathsf{fs}.id, \mathsf{fs.status} = \mathsf{clean} \vee \mathsf{dirty}) \\ \mathsf{closed} \Rightarrow & \exists\ \mathsf{fs.\ isValidClosed(fs)} * \ell \mapsto (\mathsf{fs}.id, \mathsf{fs.status} = \mathsf{closed}) \end{array} \right\}$$

which is simply $\varphi_{\mathsf{distributedfile}}$ the state to-flush removed. Then, the interesting case for the proof would be

$$(\mathsf{opened}, \mathsf{T}) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (\mathsf{to-flush}, \mathsf{T}) \rightarrow \varphi_{\mathsf{distributedfile}}(\mathsf{to-flush}) \vdash \varphi_{\mathsf{opened}}(\mathsf{opened})$$

Having isValid weaker than isValidDirty makes

$$\varphi_{\mathsf{distributedfile}}(\mathsf{to-flush}) \vdash \varphi_{\mathsf{opened}}(\mathsf{opened})$$

proven.

5.3 Invariants against Guarantee-Step Bisimulation

Unlike what we discuss in Section 4.3 for Theorem Guarantee Bisim 4.3.1 where the bisimulation relation for the guarantee steps does not *consider invariants*, here with the law of Guarantee in Figure 5.1, we incorporate the invariants to the bisimulation relation. We need to ensure that actions that

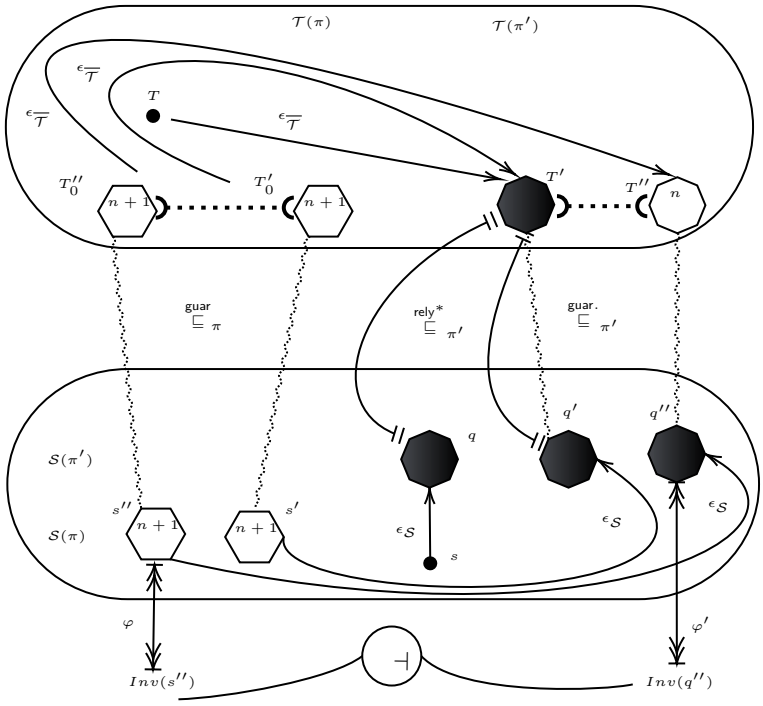


Figure 5.4 Invariants against Guarantee-Step Bisimulation

establish a certain postcondition for a target STS step also establish a corresponding postcondition for a source STS step. The law of Gurantee asserts that the corresponding local steps ensures that for every possible transition in the submodel guarantee, its postcondition implies the existence of a corresponding transition in the original which we already discussed in depth in Section 4.3 for Theorem Guarantee Bisim. The correspondence of the guarantee steps is valid with respect to the *starting state* in the *target* which is determined by the rely relation considering the plausible client interference

$$(q; T') \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (q'; T')$$

in the law of Guarantee in Figure 5.1.

Regarding the invariants, the law of Guarantee asserts that the invariant holding at the post state (shown as s'' in Figure 5.4) is implied by the post-state on the target machine (shown as q'' in Figure 5.4).

CHAPTER 6

PROGRAM LOGIC

The motivation for bisimulation in modal logic is to describe the cases where two Kripke models are indistinguishable by any modal formula. Here, our goal is similar: two STSes in bisimulation should be indistinguishable within the program logic: programs verifiable against one should be verifiable against the other. This chapter introduces the logical machinery required to incorporate the bisimulation relation explained in Chapters 5 and 4 into a program logic so that we can ensure that a client's correctness is *indistinguishable* with respect to two specifications encoded as state machines that are bisimilar to each other via the bisimulation relation shown in Figure 5.1.

6.1 Soundness of Invariants

Before explaining how we integrate the bisimulation relation with a program logic, specifically Iris, it is worth explaining how we take a step in the specifications encoded as STSes, i.e., Rule UpdIsl briefly mentioned in Section 2.1.

UpdIsl

$$\frac{\alpha \text{ physically atomic} \qquad \forall s_0 \,.\, ((s;\mathsf{T}) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (s_0;\mathsf{T})) \vdash \{\varphi(s_0) * P\}\ \alpha\ \{\exists s' \,,\, \mathsf{T}' \,.\, (s_0;\mathsf{T}) \overset{\mathsf{guar}^*}{\sqsubseteq}_{\pi} (s';\mathsf{T}') * \varphi(s') * Q\}}{\boxed{\varphi}^{\gamma}_{\pi} \vdash \{\boxed{s;\mathsf{T}}^{\gamma} * P\}\ \alpha\ \{\exists\, s', \mathsf{T}'. \boxed{s';\mathsf{T}'}^{\gamma} * Q\}}$$

The rule UpdIsl asserts the validity of any client action α specificied with the state machine π such that the behaviour of atomic action is allowed with respect to the client interference at the state s with the capabilities T. The client action must comply with the precondition for any rely reachable

state(s_0)

$$\forall s_0 \,.\, (s;\mathsf{T}) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (s_0;\mathsf{T})$$

and the post-condition requires the existence of guarantee reachable state (s') from any state the rely reachable state in the pre-condition. Then, the invariant of s' needs to be establised.

Island Update and Validity of Bisimulation Relation The island update rule is the main way to validate programs advancing an STS, so one way to let clients of one STS use another is to translate their proofs, and that finding a way to translate uses of island update is one way to do that. We prove the validity of these theorems within the bisimulation relation in Figure 5.1 against the rule UpdIsl. In Island Update Invariance 6.1.1, we want to prove, abstractly, that any island update performed by Rule UpdIsl in terms of the target STS (π') is valid in terms of the original STS (π). Intuitively, we'd like to think of an update in terms of the sub-STS as equivalent to a combination of consequence, and island update on the original. In terms of the theorem stated, considering any state q that embeds the initial state of bisimulation, $\epsilon_{\mathcal{S}}(s) = q$, if there exist states reachable via arbitrary interference and a single step on the target machine π', shown as q' and q'', respectively, then there exists a corresponding sequence of interference plus a single step in the source machine π.

Theorem 6.1.1 (Island Update Invariance)

$$\begin{aligned}
&\forall\, \pi\,, \pi'\,, \varphi\,, \varphi'\,, s\,, \mathsf{T}\,, \mathsf{U}\; (\sim : \mathcal{M}\ \pi\ \pi'\ \varphi\ \varphi' s \mathsf{T}\ \mathsf{U}) \\
&\qquad \forall\, q'\,, q''\ \mathsf{T}''.\, (\epsilon_{\mathcal{S}}(s); \epsilon_{\overline{\mathcal{T}}}(\mathsf{T})) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (q'; \epsilon_{\overline{\mathcal{T}}}(\mathsf{T})) \rightarrow \\
&\qquad\qquad (q'; \epsilon_{\overline{\mathcal{T}}}(\mathsf{T})) \overset{\mathsf{guar}}{\sqsubseteq}_{\pi'} (q''; \epsilon_{\overline{\mathcal{T}}}(\mathsf{T}'')) \wedge \\
&\qquad\qquad \exists\, s'\ \mathsf{T}_0.\, (s;\mathsf{T}) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (s';\mathsf{T}) \wedge \\
&\qquad\qquad\qquad \exists\, s''\,, \mathsf{T}_1.\, (s';\mathsf{T}_0) \overset{\mathsf{guar}}{\sqsubseteq}_{\pi} (s'';\mathsf{T}_1) \wedge \epsilon_{\mathcal{S}}(s'') = q'' \wedge \mathsf{T}_1 \equiv \mathsf{T}''
\end{aligned}$$

Proof: Specializing $\sim_{\mathsf{Rely}}$ and specializing the right-to-left direction of the rely law with the assumption

$$(\epsilon_{\mathcal{S}}(s); \epsilon_{\overline{\mathcal{T}}}(\mathsf{T})) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (q'; \epsilon_{\overline{\mathcal{T}}}(\mathsf{T}))$$

proves the goal

$$\exists\, s'\ \mathsf{T}_0.\, (s;\mathsf{T}) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (s';\mathsf{T})$$

with required existentials.

$$\exists\, s''\ , \mathsf{T}_1.\, (s';\mathsf{T}_0) \overset{\mathsf{guar}}{\sqsubseteq}_{\pi} (s'';\mathsf{T}_1)$$

follows from proper specialization of $\sim_{\mathsf{Guarantee}}$ of $\mathcal{M}$. □

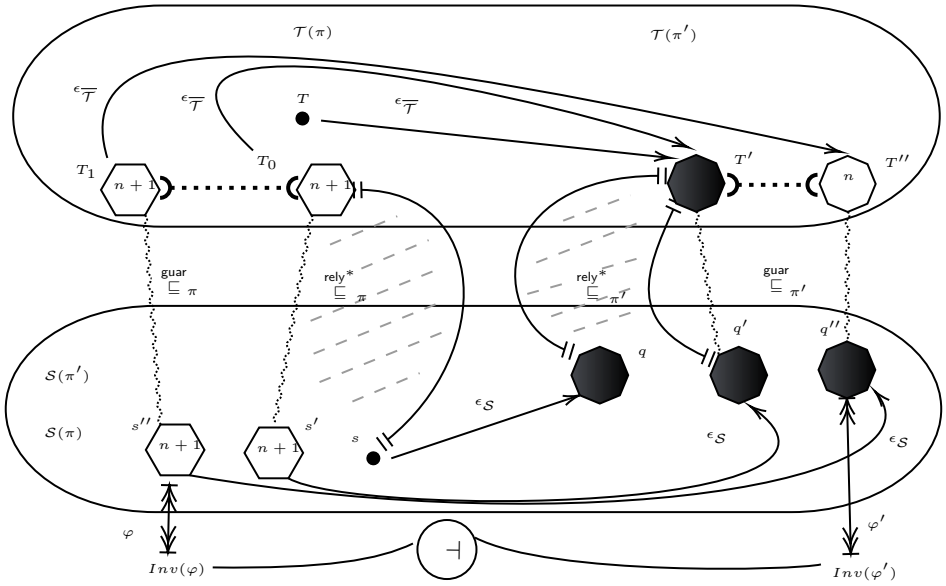


Figure 6.1 Soundness of Bisimulation against Rule UpdIsl

Matching rely steps abstracts the admissible client interference on each state machine (e.g. $\boxed{s;\mathsf{T}}^{\gamma}$), i.e. the piece of specification changed by Rule UpdIsl. In fact, Theorem Island Update Invariance 6.1.1 asserts the valid *transfer* of rely-steps, in other words *frames* in the proof rules, such that once this validity satisfied, we have the right context to transfer the proof knowledge for the local steps, i.e. matching of the guarantee steps. Looking at the proof of Theorem Island Update Invariance 6.1.1 shown in Figure 6.1 pictorially, we see the frames – transferred rely-steps – as rely surfaces (dashed surfaces in Figure 6.1) between s and s' in the source state machine, and between q and q' in the target state machine π' in Figure 6.1. These rely surfaces ensure the validity of corresponding local steps taken which are shown as guarantee surfaces between s' and s'' in the source state machine π, and between q' and q'' in the target state machine π' whose corresponding start and end states are

bisimulated.

6.2 An STS Aware Client Specfication

At this point, after understanding the validity of updates on the bisimulated protocols (designed as STSes) used to specify a program action, we can utilize this understanding in transferring a *known* proof for a program specified with an STS to the verification of the same program specified with another *bisimilar* STS. In particular, we construct a client proof specified with the source state machine out of the proof of the client specified with the target state machine that is bisimilar to the source state machine.

Our approach is to first associate specifications using bisimilar state machines with a subset of program actions in Iris HeapLang, excluding function and recursion. Since Iris relates specifications with the program actions to the weakest-precondition

$$\mathsf{P} \;-\!\!*\; \mathsf{WP}\ e\ @\ \mathsf{S}\ ;\ \mathsf{E}\{\{\ \mathsf{Q}\ \}\}$$

with the pre-condition P and the post-condition specifying the program state after executing the program action e. Our proof rule would have the form

$$\{\ \mathsf{P} * \mathsf{Pst}_{\mathsf{target}} * \mathsf{R}\ \} \;-\!\!*\; \mathsf{WP}\ e\ @\ \mathsf{S}\ ;\ \mathsf{E}\{\{\ \mathsf{Q} * \mathsf{Qst}_{\mathsf{target}} * \mathsf{R}\ \}\} \vdash$$
$$\{\ \mathsf{P} * \mathsf{Pst}_{\mathsf{source}} * \mathsf{R}\ \} \;-\!\!*\; \mathsf{WP}\ e\ @\ \mathsf{S}\ ;\ \mathsf{E}\{\{\ \mathsf{Q} * \mathsf{Qst}_{\mathsf{source}} * \mathsf{R}\ \}\}$$

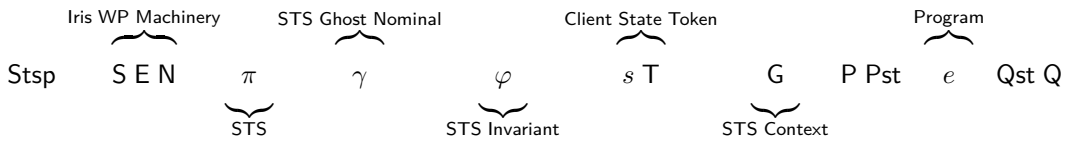
Iris weakest-preconditions themselves are ordinary *propositions*, consequently, if we encode the proof rule for *transferring a proof with target state machine to a proof with source state machine* in terms of Iris weakest-preconditions, we cannot treat the proof referring to the target state machine

$$\{\ \mathsf{P} * \mathsf{Pst}_{\mathsf{target}} * \mathsf{R}\ \} \;-\!\!*\; \mathsf{WP}\ e\ @\ \mathsf{S}\ ;\ \mathsf{E}\{\{\ \mathsf{Q} * \mathsf{Qst}_{\mathsf{target}} * \mathsf{R}\ \}\}$$

as a data to be utilized to construct the proof with the source state machine because Iris weakest-

preconditions are encoded as this implication, which cannot be directly inspected in Coq. So, with our specifications, we need to be able to treat specifications as data structures for another proof construction.

To do so, we introduce an inductive specification definition, Stsp



which bookkeeps STS relavant specification exclusively. Concretely speaking, we can think of Stsp as Iris weakest-precondition

$$\mathsf{P} \twoheadrightarrow \mathsf{WP}\ e\ @\ S\ E\ \{\{v,\ \mathsf{Q}\ v\}\}$$

decorated with a state machine name (π), state interpratation function (φ), a nominal for the ownership of the token resources of the client (γ), the current state (s) and token capabilities (T) that are used to refer to the state machine π from the client view, and most notably

- G: which is the Iris persistent predicate context made explicit on Stsp for $\boxed{\varphi}^{\gamma}_{\pi}$
- Pst: which is the pre-condition bookkeeping the client token ownership – $\boxed{s; \mathsf{T}}^{\gamma}$
- Qst: which is the post-condition bookkeeping the client token ownership

Explicit Bookkeeping of STS Relevant Specification We specify a subset of Iris HeapLang program actions in Figure 6.2. Some of these rules such as the rule of consequence (Stsp_conseq), the rule of frame (Stsp_frame and the rule for island-update (Stsp_island) requires explicit bookkeeping of how STS related pieces of the specification (G, Pst, and Qst) should behave structurally for the rules frame and consequence, and behaviourly against the execution of the program execution. The rule for island update requires the knowledge

$$\left(\begin{array}{l} \forall\ s0\ .\ (s, \mathsf{T})\ \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi}\ (s0; \mathsf{T}) \twoheadrightarrow (\varphi(s0)) * \mathsf{P} \twoheadrightarrow \\ (\mathsf{WP}\ e\ @\ S\ ;\ (\mathsf{E} \setminus \uparrow \mathsf{N})\{\{\ v\ ,\ \mathsf{Q}\ v * \exists\ (s'\ , T'\ .\ \ulcorner (s0; \mathsf{T})\ \overset{\mathsf{guar}^*}{\sqsubseteq}_{\pi}\ (s'; \mathsf{T}')\urcorner * (\varphi(s'))\}\})) \end{array}\right)$$

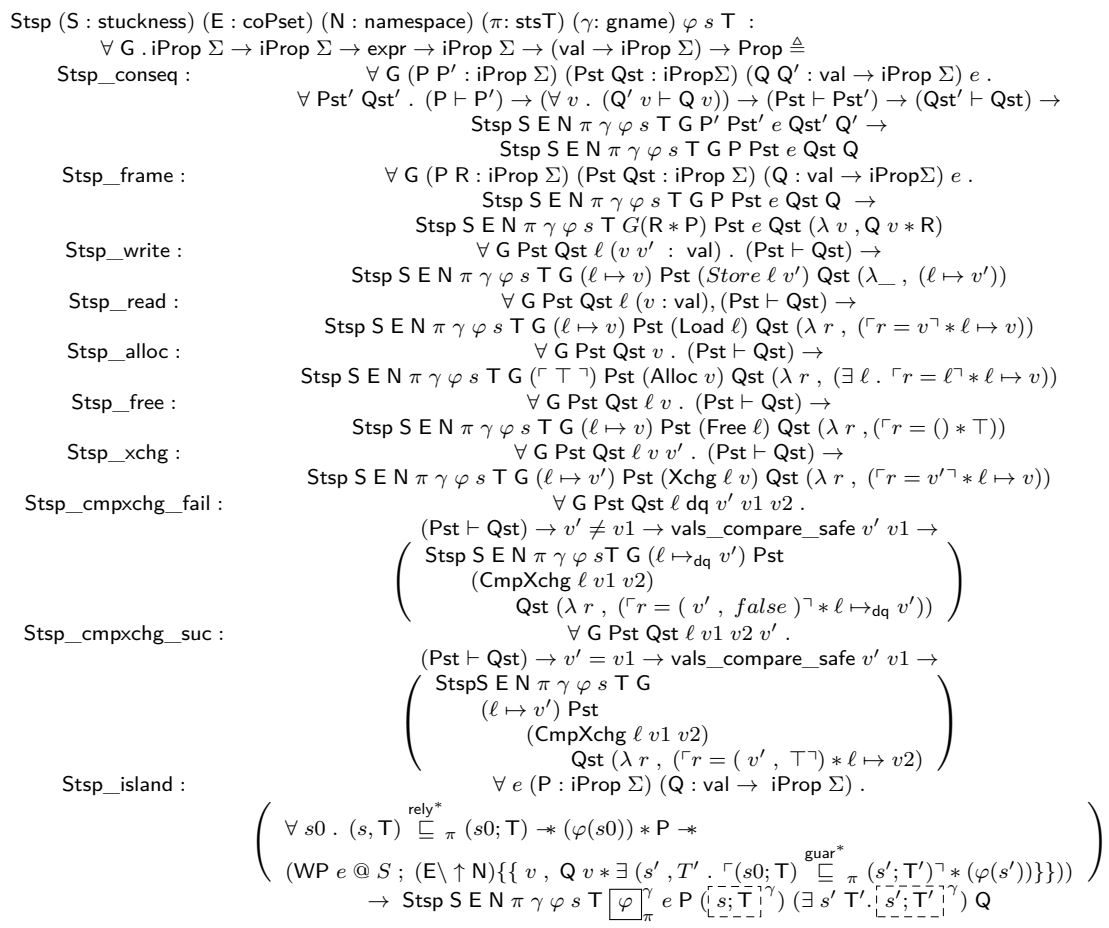


Figure 6.2 The Definition of Stsp for a Set of Program Actions in Iris HeapLang

on how we change the client's view of STS in the precondition Pst $\triangleq$ ($\boxed{s;\mathsf{T}}^{\gamma}$) at the state s' (a locally reachable) satisfying φ. The frame and consequence rules do not only change the ordinary pre- and post-conditions, P and Q, respectively, but also Pst and Qst structurally by *weakening* and *framing*.

Soundness of Stsp As mentioned, Stsp is a decorated weakest precondition of Iris. In other words Stsp can be restated in terms of Iris weakest-precondition – Theorem Stsp Soundness 6.2.1.

Theorem 6.2.1 (Stsp Soundness)

$$\forall\ \mathsf{S}\ \mathsf{E}\ \mathsf{N}\ \pi\ \gamma\ \varphi\ s\ \mathsf{T}\ \mathsf{G}\ \mathsf{P}\ \mathsf{Pst}\ e\ \mathsf{Qst}\ \mathsf{Q}.$$
$$\ulcorner \uparrow N \subseteq \mathsf{E} \urcorner \mathrel{-\!\!*} \ulcorner \mathsf{Stsp}\ \mathsf{S}\ \mathsf{E}\ \mathsf{N}\ \pi\ \gamma\ \varphi\ s\ \mathsf{T}\ \mathsf{G}\ \mathsf{P}\ \mathsf{Pst}\ e\ \mathsf{Qst}\ \mathsf{Q} \urcorner \mathrel{-\!\!*}$$
$$(\Box\ \mathsf{G} * \mathsf{P}) * \mathsf{Pst} \mathrel{-\!\!*} \mathsf{WP}\ e\ @\ S\ ;\ \mathsf{E}\{\{\ v\ ,\ \mathsf{Q}\ v * \mathsf{Qst}\}\}$$

Proof: Induction on

$$\mathsf{Stsp}\ \mathsf{S}\ \mathsf{E}\ \mathsf{N}\ \pi\ \gamma\ \varphi\ s\ \mathsf{T}\ \mathsf{G}\ \mathsf{P}\ \mathsf{Pst}\ e\ \mathsf{Qst}\ \mathsf{Q}$$

leaves us to prove the weaksest precondition for each of Stsp constructors. Most cases follow directly from the inductive hypotheses, and the interesting ones that are worth discussing would be

- **Case-Consequence**. Induction leaves us with the inductive hypothesis

$$\mathsf{IH}: \quad \mathsf{Stsp}\ \mathsf{S}\ \mathsf{E}\ \mathsf{N}\ \pi\ \gamma\ \varphi\ s\ \mathsf{T}\ \mathsf{G}\ \mathsf{P}'\ \mathsf{Pst}'\ e\ \mathsf{Qst}'\ \mathsf{Q}'$$

  which we can apply to the goal after application of weakest precondition weakening (wp_mono).

- **Case-Island-Update**. Application of Rule UpdIsl.

□

6.3 Proof Rules

At this point, we have enough logical machinery we need to construct a proof

$$\mathsf{Stsp}\ \mathsf{S}\ \mathsf{EN}\ \theta\ \gamma\ \varphi\ s\ T\ \boxed{\varphi}^{\gamma}_{\pi}\ \mathsf{P}\ \boxed{s; \mathsf{T}}^{\gamma}\ e\ (\exists\ s'\ ,\ \mathsf{T}'\ .\ \boxed{s'; \mathsf{T}'}^{\gamma})\ \mathsf{Q}$$

of a program (e) using a state macine – source state machine π – out of another proof for the same program

$$\mathsf{Stsp}\ \mathsf{S}\ \mathsf{EN}\ \theta\ \gamma\ \varphi'\ \sim .\epsilon_{\mathcal{S}}(s)\ \sim .\epsilon_{\overline{\mathcal{T}}}(T)\ \boxed{\varphi'}^{\gamma}_{\theta}\ \mathsf{P}\ \boxed{\sim .\epsilon_{\mathcal{S}}(s); \sim .\epsilon_{\overline{\mathcal{T}}}(\mathsf{T})}^{\gamma}\ e\ (\exists\ s'\ ,\ \mathsf{T}'\ .\ \boxed{s'; \mathsf{T}'}^{\gamma})\ \mathsf{Q}$$

using another state machine – target state machine θ – that is bisimlar to the source state machine. We present a proof rule Theorem Stsp Bisim 6.3.1 which considers the proof transfer through the bisimulation of source and target machines.

Theorem 6.3.1 (Stsp Bisim)

$$\forall \gamma\ \pi\ \theta\ \varphi\ \varphi'\ \mathsf{N}\ \mathsf{T}\ \mathsf{U}\ \mathsf{S}\ \mathsf{E}\ s\ \mathsf{G}\ \mathsf{P}\ e\ \mathsf{Q}$$

$$(\sim\!: \mathcal{M}\ \pi\ \theta\ \varphi\ \varphi'\ s\ \mathsf{T}\ \mathsf{U}).$$

$$\mathsf{Stsp}\ \mathsf{S}\ \mathsf{E}\ \mathsf{N}\ \theta\ \gamma\ \varphi'\ \sim.\epsilon_{\mathcal{S}}(s)\ \sim.\epsilon_{\overline{\mathcal{T}}}(T)\ \boxed{\varphi'}^{\gamma}_{\theta}\ \mathsf{P}\ \boxed{\sim.\epsilon_{\mathcal{S}}(s); \sim.\epsilon_{\overline{\mathcal{T}}}(\mathsf{T})}^{\gamma}\ e\ (\exists\ s'\ ,\ \mathsf{T}'\ .\ \boxed{s'; \mathsf{T}'}^{\gamma})\ \mathsf{Q}\ \rightarrow$$

$$\mathsf{Stsp}\ \mathsf{S}\ \mathsf{E}\ \mathsf{N}\ \pi\ \gamma\ \varphi\ s\ T\ \boxed{\varphi}^{\gamma}_{\pi}\ \mathsf{P}\ \boxed{s; \mathsf{T}}^{\gamma}\ e\ (\exists\ s'\ ,\ \mathsf{T}'\ .\ \boxed{s'; \mathsf{T}'}^{\gamma})\ \mathsf{Q}$$

Proof: We start with the proof context including the proof of the program with the target state machine ($\mathsf{IH}\theta$) and, the input of knowledge needed for utilizing the bisimulation relation (BisimInput)

$$\mathsf{IH}\theta:\ \ \mathsf{Stsp}\ \mathsf{S}\ \mathsf{E}\ \mathsf{N}\ \theta\ \gamma\ \varphi'\ \sim.\epsilon_{\mathcal{S}}(s)\ \sim.\epsilon_{\overline{\mathcal{T}}}(T)\ \boxed{\varphi'}^{\gamma}_{\theta}\ \mathsf{P}\ \boxed{\sim.\epsilon_{\mathcal{S}}(s); \sim.\epsilon_{\overline{\mathcal{T}}}(\mathsf{T})}^{\gamma}\ e\ (\exists\ s'\ ,\ \mathsf{T}'\ .\ \boxed{s'; \mathsf{T}'}^{\gamma})\ \mathsf{Q}$$

Induction on the $\mathsf{IH}\theta$ would leave us with the proof context with the target-proof constructed for each of the program actions and the proof goal as the proof to be constructed for source-state machine for the relevant case. All the cases are trivial by application of the relevant proof constructor, e.g., Stsp_frame in Figure 6.2. The interesting case in the proof is for the island update case follows from applying the Stsp constructor for island update, then applying the island update invariance theorem (Island Update Invariance 6.1.1) to transfer the assumptions from the target to the source.

□

6.4 Transferring the Proof of a File Protocol Client

Regarding the file write operation with respect to our traditional and distributed file protocols shown in Figure 4.3. Knowing the proof of *write* operation writing the value vn to the file ℓ against the

traditional file protocol

$$\mathsf{Stsp\ S\ E\ N}\ \gamma\ \varphi_{\mathsf{file}}\ (\ell\ \mapsto\ \mathsf{vp})\ \epsilon_{\mathcal{S}}(\mathsf{opened})\ \epsilon_{\overline{\mathcal{T}}}(\emptyset)\ \mathsf{P}\ \boxed{\epsilon_{\mathcal{S}}(\mathsf{opened}); \epsilon_{\overline{\mathcal{T}}}(\emptyset)}^{\gamma}$$

$$\mathsf{write}\ \ell\ \mathsf{vn}$$

$$(\exists\ s'\ ,\ \mathsf{T}'\ .\ \boxed{s'; \mathsf{T}'}^{\gamma})\ (\ell\ \mapsto\ \mathsf{vn})$$

we can construct the proof of the same operation against the distributed file protocol

$$\mathsf{Stsp\ S\ E\ N}\ \gamma\ \varphi_{\mathsf{distributedfile}}\ (\ell\ \mapsto\ \mathsf{vp})\ \mathsf{opened}\ \emptyset\ \mathsf{P}\ \boxed{\mathsf{opened}; \emptyset}^{\gamma}$$

$$\mathsf{write}\ \ell\ \mathsf{vn}$$

$$(\exists\ s'\ ,\ \mathsf{T}'\ .\ \boxed{s'; \mathsf{T}'}^{\gamma})\ (\ell\ \mapsto\ \mathsf{vn})$$

by our proof rule in Stsp Bisim 6.3.1.

CHAPTER 7

CONCLUSION,CONTINUING AND FUTURE WORK

7.1 Continuing Work

Extending the Support for Stsp This work currently goes in the direction of extending our Stsp structure to support more HeapLang actions – light-weight Iris integration. However, we also would like to explore more directions in Iris integration.

Continuing Work: Restricted Submodels Our intention is to exploit restricted submodel whose definition (Definition Restricted Submodel) we mention in Section 2.3. In fact, we already made our first attempt to give a bisimulation definition whose laws are concerned with *witheld tokens* in Figure 7.1.

$$
\mathcal{M}(\pi,\pi',\varphi,\varphi',s,\mathsf{T},\mathsf{U}) = \begin{cases}
\epsilon_{\mathcal{S}}: & \mathcal{S}(\pi) \mapsto \mathcal{S}(\pi') \\
\epsilon_{\mathcal{T}}: & \mathcal{T}(\pi) \mapsto \mathcal{T}(\pi') \\
\mathsf{H}: & U \subseteq T \\
\sim_{\mathsf{Rely}}: & \forall\ s'\ .\ (s\ \mathsf{T}) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (s';\mathsf{T}) \leftrightarrow \\
 & (\epsilon_{\mathcal{S}}(s);\epsilon_{\overline{\mathcal{T}}}(\mathsf{T}\setminus\mathsf{U})) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (\epsilon_{\mathcal{S}}(s');\epsilon_{\overline{\mathcal{T}}}(\mathsf{T}\setminus\mathsf{U})) \\
\sim_{\mathsf{Guarantee}}: & \forall\ q'\ .\ (\epsilon_{\mathcal{S}}(s);\epsilon_{\overline{\mathcal{T}}}(\mathsf{T}\setminus\mathsf{U})) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi'} (\epsilon_{\mathcal{S}}(q');\epsilon_{\overline{\mathcal{T}}}(\mathsf{T}\setminus\mathsf{U})) \rightarrow \\
 & \qquad \forall\ q'',\ \mathsf{T}''.\ (q';\epsilon_{\overline{\mathcal{T}}}(\mathsf{T}\setminus\mathsf{U})) \overset{\mathsf{guar}^*}{\sqsubseteq}_{\pi'} (q'';\mathsf{T}'') \rightarrow \\
 & \qquad \exists\ s'\ s''\ \mathsf{T0}'\ \mathsf{T0}''.(s';\mathsf{T0}') \overset{\mathsf{guar}^*}{\sqsubseteq}_{\pi} (s'';\mathsf{T0}'') \wedge \\
 & \qquad\qquad \epsilon_{\mathcal{S}}(s') = q'\ \wedge\ \epsilon_{\mathcal{S}}(s'') = q'' \wedge \\
 & \qquad\qquad (\epsilon_{\overline{\mathcal{T}}}(\mathsf{T0}'\setminus\mathsf{U})) = (\epsilon_{\overline{\mathcal{T}}}(\mathsf{T}\setminus\mathsf{U})) \wedge \\
 & \qquad\qquad (\epsilon_{\mathcal{T}}(\mathsf{T0}'\setminus\mathsf{U})) = \mathsf{T}''\ \wedge\ \varphi'(q'') \vdash \varphi(s'') \\
\sim_{\mathsf{Tolerance}} & \forall\ s'\ .\ (s\ \mathsf{T}) \overset{\mathsf{rely}^*}{\sqsubseteq}_{\pi} (s';\mathsf{T}) \leftrightarrow \varphi(s')\ \vdash\ \varphi'(\epsilon_{\mathcal{S}}(s'))
\end{cases}
$$

Figure 7.1 Bisimulation Relation Concerned with Witheld Tokens

By using this bisimulation relation, we want prove the proof rule (Theorem Stsp BisimSub 7.1.1) which considers the existence of the set of witheld tokens U. To speak concretely, assume a state machine specifying a stack library – π_{stack}. Restricting π_{stack} with respect to the `push` token yields an STS without any transitions corresponding to pushing — an STS for a pop-only stack. Thus the `push` token can (in principle) be framed away, and the stack used with code (for example, passed to procedures) verified against a smaller STS with less interference — one with stronger postconditions than one verified agains the original stack STS, but requiring fewer capabilities to invoke. The negative consequence of this is that naïvely the internal assertions of the "subtyped" code may not be stable against the interference of the "full" STS.

Theorem 7.1.1 (Stsp BisimSub)

$$
\begin{array}{l}
\forall \gamma\ \pi\ \theta\ \varphi\ \varphi'\ \mathsf{N}\ \mathsf{T}\ \mathsf{U}\ \mathsf{S}\ \mathsf{E}\ s\ \mathsf{G}\ \mathsf{P}\ e\ \mathsf{Q} \\
(\sim\!\!: \mathcal{M}\ \pi\ \theta\ \varphi\ \varphi'\ s\ \mathsf{T}\ \mathsf{U}). \\
\quad \mathsf{Stsp}\ \mathsf{S}\ \mathsf{EN}\ \theta\ \gamma\ \varphi'\ \sim .\epsilon_{\mathcal{S}}(s)\ \sim .\epsilon_{\overline{\mathcal{T}}}(T)\ \boxed{\varphi'}^{\gamma}_{\theta}\ \mathsf{P}\ \boxed{\sim .\epsilon_{\mathcal{S}}(s); \sim .\epsilon_{\overline{\mathcal{T}}}(\mathsf{T})}^{\gamma}\ e\ (\exists\ s'\ ,\ \mathsf{T}'\ .\ \boxed{s'; \mathsf{T}'}^{\gamma})\ \mathsf{Q}\ \rightarrow \\
\qquad \mathsf{Stsp}\ \mathsf{S}\ \mathsf{EN}\ \theta\ \gamma\ \varphi\ s\ T\ \boxed{\varphi}^{\gamma}_{\pi}\ \mathsf{P}\ \boxed{s; \mathsf{T}}^{\gamma}\ e\ (\exists\ s'\ ,\ \mathsf{T}'\ .\ \boxed{s'; \mathsf{T}'}^{\gamma})\ \mathsf{Q}
\end{array}
$$

7.2 Future Work

More Experiments In the future more examples should be explored to better test the limits of the approach or something like that.

Generalized Specifications RG-STS specifications use STSes which themselves are partial commutative monoids (PCM). We would like to explore whether we can generalize the principles in this thesis to a generalized concept of specification using (PCM)s. The bisimulation relation we present in this thesis is a particular form of homomorphism, but the generalization effort requires more exploration of the specification mapping than the bisimulation relation presented, e.g., *homomorphisms* between Kripke models. Once we achieved generalization, we would be able to apply the generalized version of the principles founded in this thesis to a wider range of programs with specifications using

different PCMs.

Part IV

Modal Concurrent Memory Management

CHAPTER 1

BACKGROUND

In this chapter we recall the general concepts of read-copy-update concurrency. We use the RCU linked-list-based bag[122] in Listings 1.1 and 1.2 as a running example. It includes annotations for our type system, which will be explained in Section 3.2.

```
1  struct BagNode{
2    int data;
3    BagNode<rcuItr> Next;
4  }
5  BagNode<rcuRoot> head;
6  void add(int toAdd){
7  WriteBegin;
8  BagNode nw = new;
9  {nw: rcuFresh{}}
10 nw.data = toAdd;
11 {head: rcuRoot, par: undef, cur: undef}
12 BagNode<rcuItr> par,cur = head;
13 {head: rcuRoot, par: rcultrϵ{}}
14 {cur: rcultr ϵ {}}
15 cur = par.Next;
16 {cur: rcultr Next {}}
```

```
17 {par: rcultr ϵ {Next ↦ cur}}
18 while(cur.Next != null){
19   {cur: rcultr (Next)^k.Next {}}
20   {par: rcultr (Next)^k {Next ↦ cur}}
21   par = cur;
22   cur = par.Next;
23   {cur: rcultr (Next)^k.Next.Next {}}
24   {par: rcultr (Next)^k.Next {Next ↦ cur}}
25 }
26 {nw: rcuFresh {}}
27 {cur: rcultr (Next)^k.Next {Next ↦ null}}
28 {par: rcultr (Next)^k {Next ↦ cur}}
29 nw.Next= null;
30 {nw: rcuFresh {Next ↦ null}}
31 {cur: rcultr (Next)^k.Next {Next ↦ null}}
32 cur.Next=nw;
33 {nw: rcultr (Next)^k.Next.Next {Next ↦ null}}
34 {cur: rcultr (Next)^k.Next {Next ↦ nw}}
35 WriteEnd;
36 }
```

Listing 1.1 RCU client-Add: singly linked list based bag implementation.

```
1 void remove(int toDel){
2 WriteBegin;
3 {head: rcuRoot, par : undef, cur: undef}
4 BagNode<rcuItr> par,cur = head;
5 {head: rcuRoot, par: rcultr ϵ {}, cur: rcultr ϵ {}}
```

```
cur = par.Next;
```
$\{cur$: rcultr $Next\ \{\}\}$

$\{par$: rcultr $\epsilon\ \{Next \mapsto cur\}\}$

```
while(cur.Next != null&&cur.data != toDel)
{
```
  $\{cur$: rcultr $(Next)^k.Next\ \{\}\}$

  $\{par$: rcultr $(Next)^k\ \{Next \mapsto cur\}\}$

```
  par = cur;
  cur = par.Next;
```
  $\{cur$: rcultr $(Next)^k.Next.Next\ \{\}\}$

  $\{par$: rcultr $(Next)^k.Next\ \{Next \mapsto cur\}\}$

```
}
```
$\{nw$: rcuFresh$\{\}\}$

$\{par$: rcultr $(Next)^k\ \{Next \mapsto cur\}\}$

$\{cur$: rcultr $(Next)^k.Next\ \{\}\}$

```
BagNode<rcuItr> curl = cur.Next;
```
$\{cur$: rcultr $(Next)^k.Next\ \{Next \mapsto curl\}\}$

$\{curl$: rcultr $(Next)^k.Next.Next\ \{\}\}$

```
par.Next = curl;
```
$\{par$: rcultr $(Next)^k\ \{Next \mapsto curl\}\}$

$\{cur$: unlinked$\}$

$\{cur$: rcultr $(Next)^k.Next\ \{\}\}$

```
SyncStart;
SyncStop;
```
$\{cur$: freeable$\}$

```
Free(cur);
```
$\{cur$: undef$\}$

```
33 WriteEnd;
34 }
```

Listing 1.2 RCU client-Remove: singly linked list based bag implementation.

As with concrete RCU implementations, we assume threads operating on a structure are either performing read-only traversals of the structure — *reader threads* — or are performing an update — *writer threads* — similar to the use of many-reader single-writer reader-writer locks.[1] It differs, however, in that readers may execute concurrently with the (single) writer.

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

Figure 1.1 and Figure 1.2 gives the code for adding and removing nodes from a bag respectively. Type checking for all code, including membership queries for bag, can be found in our technical report [104] Appendix D. Algorithmically, this code is nearly the same as any sequential implementation. There are only two differences. First, the read-side critical section in `member` is indicated by the use of ReadBegin and ReadEnd; the write-side critical section is between WriteBegin and WriteEnd. Second, rather than immediately reclaiming the memory for the unlinked node, `remove` calls SyncStart to begin a *grace period* — a wait for reader threads that may still hold references to unlinked nodes to finish their critical sections. SyncStop blocks execution of the 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 [48], 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

[1] RCU implementations supporting multiple concurrent writers exist [10], but are the minority.

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*[136]: 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 2 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 3 guarantees these invariants.

CHAPTER 2

SEMANTICS

In this chapter, 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[124]. In our semantics, shown in Figure 2.1, 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 2.1 gives operational semantics for *atomic* actions; conditionals, loops, and sequencing all have standard semantics, and parallel composition uses sequentially-consistent interleaving semantics.

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

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

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

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

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

Figure 2.1 Operational semantics for RCU.

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

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

CHAPTER 3

TYPE SYSTEM

In this chapter, 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 list-based bag implementation[122] in Listings 1.1 and 1.2. 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[104] Appendix C. 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.

3.1 RCU Type System for Write Critical Section

Section 3.1 introduces RCU types and the need for subtyping. Section 3.2, shows how types describe program states, through code for list-based bag example in Listings 1.1 and 1.2. Section 3.3 introduces the type system itself.

RCU Types There are six types used in Write critical sections

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

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

Read-side critical sections use 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 [104] Appendix F.

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

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

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

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

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

Subtyping It is sometimes necessary to use imprecise types — mostly for control flow joins. Our type system performs these abstractions via subtyping on individual types and full contexts, as in Figure 3.1.

Figure 3.1 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

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

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

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

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

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

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

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

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

Figure 3.1 Subtyping rules.

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

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

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

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

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

Figure 3.2 Type rules for control-flow.

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 3.1, we see that `cur` aliases different fields of `par` – either $Left$ or $Right$ – in

different branches of the conditional. The types after the conditional must overapproximate this, here as $Left|Right \mapsto cur$ in `par`'s field map, and a similar path disjunction in `cur`'s path. This is reflected in Figure 3.1's T-NSub1-5 and T-PSub1-2 – within each branch, each type is coerced to a supertype to validate the control flow join.

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

Assertions for the `add` function in lines 19 and 20 of Figure 1.1 show the *loop*'s effects on paths of iterator references used inside the loop, `cur` and `par`. On line 20, `par`'s path contains has $(Next)^k$. The k in the $(Next)^k$ abstracts the number of loop iterations run, implicitly assumed to be non-negative. The trailing $Next$ in `cur`'s path on line 19 – $(Next)^k.Next$ – expresses the relationship between `cur` and `par`: `par` is reachable from the root by following $Next$ k times, and `cur` is reachable via one additional $Next$. The types of 19 and 20, however, are not the same as lines 23 and 24, so an additional adjustment is needed for the types to become loop-invariant. *Reindexing* (T-ReIndex in Figure 3.2) 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}}
```

Listing 3.1 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 3.2) to refine imprecise paths and field maps. In Figure 3.1, 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.

3.2 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[122] from Listings 1.1 and 1.2, highlighting how the type rules handle different parts of the code. Listings 1.1 and 1.2 are 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 Listings 1.1 and 1.2 begins by reading the root into a new rcuItr reference used to begin traversing the structure. After each initial read (line 12 of add and line 4 of remove), the path of `cur` reference is the empty path (ϵ) and the field map is empty ($\{\}$), because it is an alias to the root, and none of its field contents are known yet.

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

Unlinking Nodes Line 24 of remove in Listing 1.2 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).

3.3 Type Rules

Figure 3.3 gives the primary type rules used in checking write-side critical section code as in Listings 1.1 and 1.2.

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

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

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

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

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

(T-WriteFH)

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

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

(T-UnlinkH)

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

(T-Replace)

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

(T-Insert)

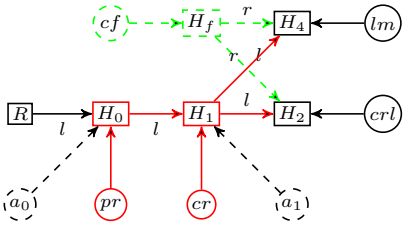
$$\frac{\begin{array}{c}\mathcal{N}' = \mathcal{N}([f \rightharpoonup o \setminus n]) \qquad \rho.f = \rho_1 \qquad \rho_1.f_4 = \rho_2 \qquad \mathcal{N}(f) = \mathcal{N}_1(f_4) \qquad \forall_{f_2 \in dom(\mathcal{N}_1)}.\, f_4 \neq f_2 \implies \mathcal{N}_1(f_2) = \mathsf{null} \\ \mathsf{FV}(\Gamma)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

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

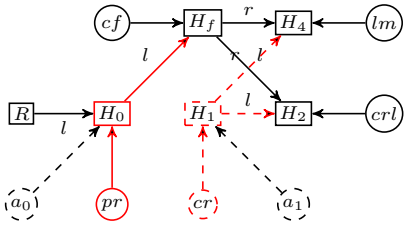
Figure 3.3 Type rules for write side critical section.

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



(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.

Figure 3.4 Replacing *existing* heap nodes with *fresh* ones. Type rule T-Replace.

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 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 3.4a and 3.4b. Our technical report [104] also includes Figures E.1a and E.1b in Appendix E to illustrate complexities in unlinking. The square R nodes are root nodes, and H nodes are general heap nodes. All resources in red and green form the memory 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 3.4a) 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 3.4a to 3.4b 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 3.3, 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 F of the technical report [104].

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 3.3 ensures unlinked references have been freed by forbidding them in the critical section's post-type environment. Our technical report [104] also includes the analogous rule ToRCURead for the read critical section in Figure F.1 of Appendix F.

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

CHAPTER 4

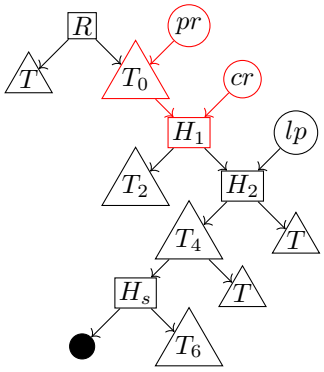
EVALUATION

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

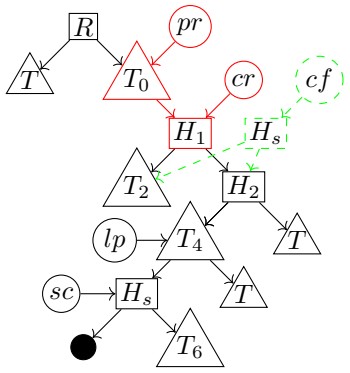
Listings 1.1 and 1.2 give the type-annotated code for add and remove operations on a linked list implementation of a bag data structure, following McKenney's example[122] respectively. Appendix D of the technical report[104] contains the code for membership checking.

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[10]: 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 C of the technical report[104], but it motivates some of the conditional-related flexibility discussed in Section 3.2. The use of disjunction $(Left|Right)$ in field maps and paths is required to capture traversals which follow different fields at different times, such as the lookup in a binary search tree.

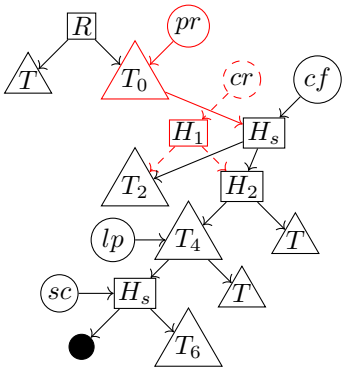
The most subtle aspect of the deletion is the final step in the case the node H_1 to remove has both children. In this case, the value H_s of the left-most node of H_1's right child — the next element in the collection order — is copied into a new *freshly-allocated* node as shown in Figure 4.1a, which is then used to *replace* node H_1 as shown in Figure 4.1c: 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 4.1b, and the parent is updated to reference the replacement, unlinking H_1. At this point, as shown in Figures 4.1c-4.1d, there are



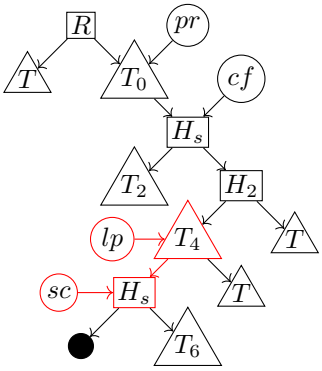
(a) The writer traverses subtree T_0 to find the heap node H_1 with local references *pr* and *cr*.



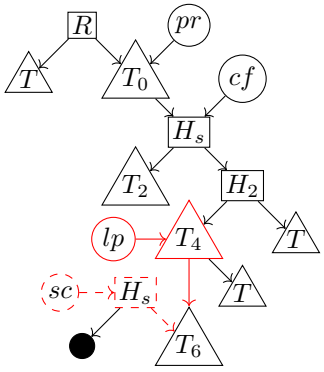
(b) Traverse subtree T_4 starting from H_2 with references *lp* and *sc* to find successor H_1' of H_1. Duplicating H_1' as a fresh heap node before replacing H_1 with the fresh one.



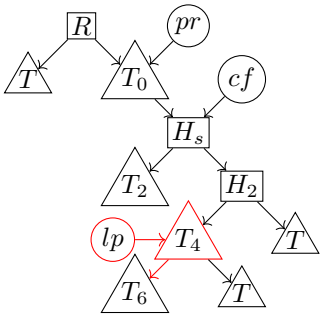
(c) Replace H_1 with fresh successor and synchronize with the readers.



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



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



(f) figure
Reclamation of the old successor.

Figure 4.1 Delete of a heap node with two children in BST[10].

two nodes with value H_s in the tree (*weak* BST property of the Citrus[10]): 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 4.1e, 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 4.1c-4.1e, 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.

4.1 Soundness

This section outlines the proof of type soundness – our full proof appears in Appendices B.1, B.2, B.3 and B.4 of the technical report [104]. We prove type soundness by embedding the type system into an abstract concurrent separation logic called the Views Framework [52], 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 [68,69], this consists of several key steps:

1. Define runtime states and semantics for the atomic actions of the language. These are exactly the semantics from Figure 2.1 in Section 2

2. Define a denotational interpretation $[\![-]\!]$ of the types (Figure 4.2) in terms of an *instrumented* execution state – a runtime state (Section 2) 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 B.2 of the technical report [104] 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 3) – 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 4.4) 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.

4.1.1 Proof

This section provides more details on how the Views Framework[52] is used to prove soundness, giving the major parameters to the framework and outlining global invariants and key lemmas.

Logical State Section 2 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 2. 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 $\mathsf{Loc} \to \mathcal{P}(\mathsf{obs})$
- 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 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.

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

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 B.2 of the technical report[104].

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

Invariants on Grace-Period Our logical state (Section 4.1.1) 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 Iterators-Free-List invariant complements this by asserting all readers with such observations are in the bounding thread set.

The writer thread can refer to a heap location in the free list with a local reference either in type

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

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

Only a bounding thread may view an (unlinked) heap location in the free list as iterator. The denotation of the reader thread's rcultr reference, $[\![\mathsf{rcultr}]\!]_{tid}$, 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 Unlinked-Reachability and Free-List-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 Fresh-Writes invariant asserts that a fresh heap location can only be allocated and referenced by the writer thread. The relation between a freshly allocated heap and the rest of the heap is established by the Fresh-Reachable invariant, which requires that there exists no heap node pointing to the freshly allocated one. This invariant supports the preservation of the tree structure. Fresh-Not-Reader invariant supports the safe traversal of the reader threads via asserting that they cannot observe a heap location as fresh. Moreover, the denotation of rcuFresh type, $[\![\mathsf{rcuFresh}\,\mathcal{N}]\!]_{tid}$, enforces that fields in $\mathcal{N}$ point to valid heap locations (observed as iterator by the writer thread).

$$
\begin{array}{lcl}
[\![\, x : \mathsf{rcultr}\,\rho\,\mathcal{N}\,]\!]_{tid} & = & \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l} (\mathsf{iterator}\, tid \in O(s(x,tid))) \wedge (x \notin U) \\ \wedge(\forall_{f_i \in dom(\mathcal{N}) x_i \in codom(\mathcal{N})} \cdot \left\{ \begin{array}{l} s(x_i,tid) = h(s(x,tid),f_i) \\ \wedge \mathsf{iterator} \in O(s(x_i,tid))) \end{array} \right. \\ \wedge(\forall_{\rho',\rho''} \cdot \rho'.\rho'' = \rho \implies \mathsf{iterator}\, tid \in O(h^*(rt,\rho'))) \\ \wedge h^*(rt,\rho) = s(x,tid) \wedge (l = tid \wedge s(x,\_) \notin dom(F))) \end{array} \right\} \\
[\![\, x : \mathsf{rcultr}\,]\!]_{tid} & = & \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l} (\mathsf{iterator}\, tid \in O(s(x,tid))) \wedge (x \notin U) \wedge \\ (tid \in B) \implies \left\{ \begin{array}{l} (\exists_{T' \subseteq B} \cdot \{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})} \cdot 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 $h^* : (\mathsf{Loc} \times \mathsf{Path}) \rightharpoonup \mathsf{Val}$

Figure 4.2 Type Environments

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

Assertions in the Views logic are (almost) sets of the logical states that satisfy a validity predicate WellFormed, outlined in Section 4.1.1:

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

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

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

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 4.2.

Individual variables' denotations are extended to context denotations slightly differently depending

on whether the environment is a reader or writer thread context: writer threads own the global lock, while readers do not:

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

Composition and Interference To support framing (weakening), the Views Framework requires that views form a partial commutative monoid under an operation $\bullet : \mathcal{M} \longrightarrow \mathcal{M} \longrightarrow \mathcal{M}$, provided as a parameter to the framework. The framework also requires an interference relation $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$ between views to reason about local updates to one view preserving validity of adjacent views (akin to the small-footprint property of separation logic). Figure 4.3 defines our composition operator and the core interference relation $\mathcal{R}_0$ — the actual inferference between views (between threads, or between a local action and framed-away state) is the reflexive transitive closure of $\mathcal{R}_0$. Composition is mostly straightforward point-wise union (threads' views may overlap) of each component. Interference bounds the interference writers and readers may inflict on each other. Notably, if a view contains the writer thread, other threads may not modify the shared portion of the heap, or release the writer lock. Other aspects of interference are natural restrictions like that threads may not modify each others' local variables. WellFormed states are closed under both composition (with another WellFormed state) and interference ($\mathcal{R}$ relates WellFormed states only to other WellFormed states).

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

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

In order to validate the frame rule in the Views Framework's logic, the assertions in its logic — sets of well-formed instrumented states — must be restricted to sets of logical states that are *stable* with

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

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

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

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

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

Figure 4.3 Composition($\bullet$) and Thread Interference Relation($\mathcal{R}_0$)

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:

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

Additionally, interference must distribute over composition:

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

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

Lemma 2 (Stable Environment Denotation-M) *For any* closed *environment* Γ *(i.e.,* $\forall x \in \mathsf{dom}(\Gamma).\, , \mathsf{FV}(\Gamma(x)) \subseteq \mathsf{dom}(\Gamma)$*):* $\mathcal{R}([\![\Gamma]\!]_{\mathsf{M},tid}) \subseteq [\![\Gamma]\!]_{\mathsf{M},tid}$*. Alternatively, we say that environment denotation is* stable *(closed under* $\mathcal{R}$*).*

Proof: In Appendix B.1 Lemma 8 of the technical report [104]. □ We elide the statement of the analogous result for the read-side critical section, available in Appendix B.1 of the technical report.

With this setup done, we we can state the connection between the Views Framework logic induced by

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

$$[\![\mathtt{assume}(\mathcal{S})]\!](s) \stackrel{\text{def}}{=} \begin{cases} \{s\} & \text{if } s \in \mathcal{S} \\ \emptyset & \text{Otherwise} \end{cases} \quad \downarrow \mathsf{while}\ (e)\ C \downarrow \stackrel{\text{def}}{=} (\mathsf{assume}(e); C)^*; (\mathsf{assume}(\neg e)); \quad \frac{\{P\} \cap \{\lceil \mathcal{S} \rceil\} \sqsubseteq \{Q\}}{\{P\}\mathtt{assume}\,(\mathcal{S})\,\{Q\}} \ \text{where}$$

$$\lceil \mathcal{S} \rceil = \{m | \lfloor m \rfloor \cap \mathcal{S} \neq \emptyset\}$$

Figure 4.4 Encoding branch conditions with assume(b)

earlier parameters, and the type system from Section 3. 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 4.4. $\mathsf{assume}(b)$ is a standard idea in verification semantics[13,128], 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 4.4 adapts this for the Views Framework as in other Views-based proofs[68,69], 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[68,69]) address subtyping in a Views-based proof. Views defines the notion of *view shift*[1] ($\sqsubseteq$) as a way to reinterpret a set of instrumented states as a new (compatible) set of instrumented

[1]This is the same notion present in later program logics like Iris[100], though more recent variants are more powerful.

states, offering a kind of logical consequence, used in a rule of consequence in the Views logic:

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

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

Lemma 3 (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 B.1 of the techical report [104]. □

Lemma 4 (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 5 (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 3 for primitives, Lemma 4 and consequence for subtyping, and otherwise appealing to structural rules of the Views logic and inductive hypotheses. Full details in Appendix B.1 of the technical report [104]. □

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.

4.2 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.[4] 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[3] satisfies this property. According to the fundamental requirements of RCU [123], our model in Section 2 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[104] 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 4.1.1.

- *Publish-Subscribe Guarantee*: Fresh heap nodes cannot be observed by the reader threads until they are published. As we see in the operational semantics, once a new heap location

is allocated it can only be referenced by a local variable of type fresh. Once published, the local type for that reference becomes 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 3.4a-3.4b) a fresh node in Section 3.3 and the related invariants in Section 4.1.1.

- *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[118] 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)[48] where synchronization between the writer thread and reader threads provided via per-thread counters. Tassarotti et al.[153] uses a protocol based program logic based on separation and ghost variables called GPS[157] 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.[63] extend Rely-Guarantee and Separation-Logic[60,61,158] 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.[70] take assertions from temporal logic to separation logic[158] to capture the essence of epoch-based memory reclamation algorithms and have a simpler proof than what Fu et al. have[63] for Michael's non-blocking stack[126] implementation under a sequentially consistent memory model.

Tassarotti et al.[153] 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 4). 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[125] 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.[95] use model-checking to test the core of Tree RCU in Linux kernel. Liang et al.[114] 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.[48] 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.[37,83] 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[83]. 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.

4.3 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.[125] 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[122] and a binary search tree[10]. 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[52] 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.

Part V

Modalities as Verification Patterns

CHAPTER 1

DEFINITIONS FOR SYSTEMS VERIFICATION PATTERNS

Although they differ in the functionality they offer, low-level systems exhibit certain patterns of design and utilization of computing resources. In this chapter, we argue the position that *modalities* should be a go-to approach when specifying and verifying low-level systems code. We explain how the concept of a *resource context* helps guide the design of new modalities for verification of systems code, and we justify our perspective by discussing prior systems that have used modalities for systems verification successfully, arguing that they fit into the verification design pattern we articulate, and explaining how this approach might apply to other systems verification challenges.

To explain our ideas in the general systems understanding, we briefly recap some of the background and themes our ideas build on again,casting them in a certain way to bring out the relevance of our philosophy.

1.1 Resources in Systems Software

Systems software, in general, interfaces with an underlying computing architecture such that any software system at any higher level in the software stack can (at least indirectly) utilize the resources of the machine. The last layer of software before the hardware is naturally critical to the correctness of an overall system, as essentially all software built on top of it assumes its correctness. And because hardware is complex and highly diverse, the implementation of those lowest layers of the software stack is typically intricate and naturally error-prone, despite how critical its correctness is. Typically

systems software has, as a primary focus, the task of *abstracting* from hardware details to simplify the construction of higher layers of the stack.

1.1.1 Virtualization

One of the most common forms of abstraction provided by systems software is *virtualization*, which abstracts the relationship between conceptual and physical computing resources. Operating System (OS) kernels virtualize memory locations and quantity (via virtual memory and paging [46]). Distributed language runtimes may virtualize addresses, even when processes may migrate across machines [88]. Filesystems virtualize locations on disk [22,82,141,142]. Programs built on top of the corresponding systems software layer work logically at the level of these virtualized resources, and it makes sense to specify the systems software directly in terms of those abstractions.

Access by Translation Accessing virtualized resources via translation is a common way to virtualize notions of location (e.g., virtual memory addresses, inodes or object IDs instead of disk addresses [22,82,141,142]). B-trees , page tables , and related structures both behave like maps, when the corresponding physical resources exist as such just in a different location. Control over the lookup process (e.g, handling the case of a missing translation entry) allows for additional flexibility, such as filling holes in sparse files, or demand paging (both from disk or lazily populating anonymous initially-zeroed mappings). Although the realization of these maps may differ from a system to system based on the context (and sometimes hardware details), they are semantically — logically — partial maps, worth treating as such in verification.

1.2 Nominals

1.2.1 Recapping Modal Operators in Program Specifications: Systems Perspective

The value of modal operators in program specifications is that they permit describing something that is true within some context(s), without needing to fully describe the context itself. Yet unlike

shorthands, abbreviations, or general definitions, modal operators interact with the logic in systematic — rather than ad hoc — ways.

For example, most modal operators M have the property that if $P \Rightarrow Q$, then $M(P) \Rightarrow M(Q)$. Many modal operators distribute over conjunction, so $M(P \wedge Q) \Rightarrow (M(P) \wedge M(Q))$. So when a program state or behaviour can be captured by a modality, this interacts nicely with the rest of the logic in ways that simplify both specifications and proofs. Here we give a few examples of how this is true in classic work; the next section argues that this is *particularly* true for systems software, because the concepts and reasoning that arise in low-level software are naturally captured by modalities.

Although we focus on understanding the modal patterns in the program logics themselves for verifying the client programs (Section 2.1), not in the logical machinery they implement, we think that it sheds light on how the modal operator utilized in the machinery of the program logics shows resemblance and can be lifted to the task of client verification.

Temporal Operators The best known class of modal logics in the systems community is undoubtedly temporal logics (whether LTL[137], CTL[58], TLA[108], or others), due in part to Lamport's influence[109] leading to its not-infrequent use in specifying distributed algorithms[2,134].

1.2.2 Nominalization

Some classes of assertions benefit specifically from *naming* the explicit conditions where they are true (as opposed to simply requiring them to be true *somewhere* or *everywhere* as in the most classic modal operators). This naming generally resembles the *satisfaction* operator of *hybrid logic*[11,18]: $@_\iota P$ which evaluates the truth of P at the named (Kripke model) state ι. For this reason we refer to the general idea of naming circumstances explicitly as *nominalization*, even though the examples we discuss are not necessarily actually hybrid logics.

An example utilization of state naming explicitly on the assertion appears in program logics such as Iris[90], which enables encoding of usage protocols (e.g. state transition system) resembling typestate[64,148] as specifications. Protocol assertions are *annotated with the name of the last (abstract)*

state at which the protocol is ensured. An example more familiar to the systems community is Halpern et al.'s history of adapting modal logics of knowledge to deal with distributed systems[75,76,78]. In most of that line of work, $\mathcal{K}_a(P)$ indicates that the node a in the system *knows* or *possesses knowledge of* P (for example, a Raft node may "know" a lower bound on the commit index). Alternatively, a modality $@_i(P)$ may represent that P is true of/at the specific node i[67] (e.g., that node i has stored a certain piece of data to reliable storage). These permit capturing specific concepts relevant to the correctness (and reasoning about correctness) of a certain class of systems, involving facts about specific named entities in the system.

In each of these cases, the fact that these facts are described using modalities with the core modal property $P \Rightarrow Q$, then $M(P) \Rightarrow M(Q)$ means, for example, that if a process p knows that the commit index is greater than 5 (e.g., $\mathcal{K}_p(\mathsf{commitIndex} > 5)$) no extra work is required to conclude that the node knows it is greater than 3 (i.e., that $\mathcal{K}_p(\mathsf{commitIndex} > 3)$), because this follows from standard properties of modal operators as described above. In contrast, if verification instead used a custom assertion $\mathsf{minCommitIndex}(5)$ to represent the former knowledge, one would need to separately provide custom reasoning to conclude $\mathsf{minCommitIndex}(3)$.

CHAPTER 2

CONTINGENCY DECOMPOSITION OF A SYSTEM

2.1 Decomposing a System into its Constituents *Contingently*

Table 2.1 Modal Decomposition of Program-Logics.

	Resource Context	Resource Elements	Nominalization	Context Steps
Post-Crash Modality [27,28,154]	$\Diamond\ P$	$\ell \mapsto^{\overline{\gamma}}_{n} v$	Strong	Crash Recovery
NextGen Modality [160]	$\overset{t}{\hookrightarrow}\ P$	Own (t(a))	Strong	Determined Based on the Model*
StackRegion Modality* [160]	$\overset{ICut^{n}}{\hookrightarrow}\ P$	$\boxed{\text{n}}\ \ell \mapsto v$	Strong	Alloc and Return to/from stack
Memory-Fence Modality [40,56,57]	$\triangle_{\pi}$ and $\triangledown_{\pi}$	$\ell \mapsto v$	Weak	Fence Acquire and Release
Address-Space Modality [106]	[r]P	$\ell \mapsto v$	Weak	Address-Space Switch
Ref-Count Modality [162]	$@_{\ell}$ P	$\ell_1 \mapsto v$	Weak	Allocating, Dropping and Sharing a Reference

*The StackRegion Modality is an instance of NextGen (called the Independence Modality in [160]).

Lots of existing program logics for system verifications have a common structure, which maps to modalities with a couple extra dimensions of design. We summarize our discussion of these logics in Table 2.1. We discuss, based on examples, common aspects of how we intuitively think about correctness of systems code in many contexts, articulate those pieces, and call out the commonalities across a range of systems.

2.2 Resource

Consider first the address-space abstraction in an OS kernel. An address-space of a process is a container of virtual addresses referencing data in memory. One would expect to have *points-to* assertion from separation logic to specify *ownership* of a memory reference pointing to some data. But that ownership is relative to a specific address space — a specific container. We tend to think

directly about what is true *in an address space*, with the simplest piece being an association between a virtual address and the data it points to. We call the simplest piece, in this and other examples, the *resource element*:

Definition 8 (Resource Elements) *The simplest atomic facts we want to work with in a particular setting, specific to that setting.*

By definition, the resource elements are specific to some limited domain or setting. For example, knowing that a certain address points to a 32-bit signed integer representing 3 is knowledge restricted to a certain address space. In general, we call these domains that any resource element is tied to *resource contexts*:

Definition 9 (Resource Context) *A resource context is an abstraction, context, or container of resource elements of the same type, e.g., an address space of a process.*

We discuss a range of examples for each of these in turn.

Table 2.1 gives additional examples of systems and their corresponding resource elements and contexts where these elements reside, though none of the work in that table analyzes itself according to the structure we are giving.

Except for Post-Crash-Modality, one can think of the resource contexts in the first column in Table 2.1 as containers for the corresponding resource elements in the second column.

Stack Regions When reasoning about stack frame contents, the resource element would be a stack-memory points-to assertion ($\boxed{\mathrm{n}}\ \ell \mapsto v$) indicating that a certain offset into stack region n holds value v.

Virtual Memory For virtual memory management, a *virtual-points-to* ownership assertion pairing a virtual address (ℓ) with data (v) in an address space is natural. A process's address space with the root address r is an abstraction that is treated as a container for virtual address mappings, $\ell \mapsto v$;

Weak-Memory When considering weak memory models, we also want points-to information (address-value mappings).

Reference Counting When dealing with reference-counting APIs, we may care to specify reachability of memory nodes ($\ell \mapsto v$) in a certain context defined by a shared root address. A shared memory address ℓ can be the root of the graph that can be a container of memory nodes ($\ell_1 \mapsto v$) that are reachable from the root ℓ.

Post-Crash The resource element of Post-Crash Modality is not obvious in Table 2.1, and needs a bit of explanation. Perennial, based on the Iris logic, has both disk-points-to assertions $d[\mathsf{p}] \mapsto_n v$ (for a specific disk d) and in-memory points-to assertions $\ell \mapsto^{\overline{\gamma}}_n$. Perennial crash-recovery logic book-keeps resource names (can be thought of as logical variables) $\overline{\gamma}$ to identify which assertions (resource elements) remain valid after a crash — these assertions are only usable while the names in $\overline{\gamma}$ are valid, and a crash resets them, discarding assumptions about volatile state. A subtlety of the notion of a resource context is that, unlike the earlier examples, the context does not need to be a literal data structure. It can instead be (various forms of) a set of executions, as in the Post-Crash and NextGen modalities. The Post-Crash modality $\diamond$ expresses that the assertion P will be true after a crash discards all unstable storage (i.e., RAM). This was the inspiration for the NextGen modality, which is in fact a framework for defining "after-t" modalities, where t is an transformation of the global state.[1]

2.3 Nominalization

Finally, another design point is the question of whether or not resource element assertions must explicitly track their corresponding context, or if they implicitly pick up their context from where they are used.

Strong nominalization is the case where resource elements must explicitly include the identity of their intended context, while *weak nominalization* occurs when the resource elements implicitly pick up

[1]The transformations are subject to some technical constraints that are unrelated to our point here.

the relevant context from how they are used. The first three modalities in Table 2.1 are strongly nominalized, with the resource elements generally carrying identifiers of a specific modality usage.[2] The last three are weakly nominalized.

This choice trades off complexity against flexibility and scoping constraints. Strongly nominalized elements track additional specifier/prover-visible book-keeping data. But in exchange for carrying those identifiers of their context with them, strongly nominalized elements can be used together under any modality. For example, one can use the StackRegion modality to talk about two different stack frames simultaneously for code which accesses multiple stack frames: $\boxed{\text{n}}\,\ell \mapsto v * \boxed{\text{n+1}}\,\ell' \mapsto v'$. Using a given strongly nominalized assertion element under different modalities for different frames does not change its meaning.

By contrast, weakly nominalized elements are more concise, but then make talking about multiple contexts together marginally more complex: changing which modality an assertion is used with drastically changes its meaning. In the case of the Ref-Count modality, $@_\ell(\ell_1 \mapsto v)$ says that ℓ points to a reference count wrapping ℓ_1, while placing the $\ell_1 \mapsto v$ under a jump modality for a different location entails talking about a different region of memory.

In general, use cases where code frequently manipulates small parts of multiple contexts together should prefer strong nominalization, while use cases where usually larger portions of a single context are at issue should prefer weak nominalization.

[2]The Post-Crash modality does not look like this in the presentation here; technically the definition of $\diamond$ quantifies over names $\overline{\gamma}$ internally, dealing with sets of possible contexts.

CHAPTER 3

CONCLUSION

3.1 Making It Work

While the choice of resource elements, contexts, ambiance, and nominalization to capture the setting and contextually-relevant pieces of information help lay out the shape of specifications based on how we would like to think about them, there are a few more steps beyond designing the specifications.

It is necessary to work out how proof rules interact with the modalities — which program operations or hardware primitives update machine state in a way that interacts with the modality? This could be all or some, depending on the use case.

It is also necessary to implement the approach. This design has been most thoroughly explored with logics embedded in proof assistants (everything in Table 2.1 uses Coq), in which case a framework like Iris[90] (used for all of our examples) will require proofs that each rule interacting with the modality is sound, but in exchange for that work and a few supplementary definitions, modal assertions can work naturally with existing tools for the logic[99]. However, it is also suitable for use with automated techniques, for example by writing new axioms corresponding to proof rules[67], though better automation may require more work.

3.2 Conclusion

We have argued the essential patterns that help to choose the modalities when working out how to specify different kinds of systems code based on recent successes. It also captures (a slightly

more organized version of) our own thinking in coming up with designs for specifying distributed systems[67], virtual memory managers[106], and in ongoing work using different modalities for different parts of a copy-on-write filesystem (e.g., for assertions true in different snapshots). So not only are the modalities useful for conducting proofs, the pieces we have identified seem to be an effective way of working out a specific modality for a specific use-case.

As our main conclusion and future work, we think that these essential patterns in the designs of modal abstractions discussed constitute the fundamentals of *verification patterns* when working out how to specify different kinds of systems code.

BIBLIOGRAPHY

[1] 2021. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 20211203*. Technical Report. https://riscv.org/technical/specifications/

[2] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2019. WPaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 211–223.

[3] Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 141–157. https://doi.org/10.1007/978-3-642-39799-8_9

[4] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 405–418. https://doi.org/10.1145/3173162.3177156

[5] Eyad Alkassar, Mark A Hillebrand, Dirk Leinenbach, Norbert W Schirmer, and Artem Starostin. 2008. The Verisoft approach to systems verification. In *Verified Software: Theories, Tools, Experiments*. Springer, 209–224.

[6] Eyad Alkassar, Wolfgang J Paul, Artem Starostin, and Alexandra Tsyban. 2010. Pervasive verification of an OS microkernel. In *Verified Software: Theories, Tools, Experiments*. Springer, 71–85.

[7] Eyad Alkassar, Norbert Schirmer, and Artem Starostin. 2008. Formal pervasive verification of a paging mechanism. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 109–123.

[8] AMD. 2023. AMD64 Architecture Programmer's Manual, Volume 2: System Programming. Revision 3.40.

[9] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 109–122. https://doi.org/10.1145/1190216.1190235

[10] Maya Arbel and Hagit Attiya. 2014. Concurrent Updates with RCU: Search Tree As an Example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC '14)*. ACM, New York, NY, USA, 196–205. https://doi.org/10.1145/2611462.2611471

[11] Carlos Areces, Patrick Blackburn, and Maarten Marx. 2001. Hybrid logics: Characterization, interpolation and complexity. *The Journal of Symbolic Logic* 66, 3 (2001), 977–1010.

[12] Mike Barnett, Bor-Yuh Evan Chang, Rob DeLine, and Bart Jacobs. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*. Springer. https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/krml160.pdf

[13] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-oriented Programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects (FMCO'05)*. Springer-Verlag, Berlin, Heidelberg, 364–387. https://doi.org/10.1007/11804192_17

[14] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and Verification: The Spec# Experience. *Commun. ACM* 54, 6 (June 2011), 81–91. https://doi.org/10.1145/1953122.1953145

[15] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification*. Springer, 171–177.

[16] Lars Birkedal and Rasmus Ejlers Møgelberg. 2013. Intensional type theory with guarded recursive types qua fixed points on universes. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*. IEEE, 213–222. https://doi.org/10.1109/LICS.2013.27

[17] Lars Birkedal, Rasmus Ejlers Mogelberg, Jan Schwinghammer, and Kristian Stovring. 2011. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*. IEEE, 55–64.

[18] Patrick Blackburn and Jerry Seligman. 1995. Hybrid languages. *Journal of Logic, Language and Information* 4, 3 (1995), 251–272.

[19] Patrick Blackburn and Johan Van Benthem. 2006. 1 Modal logic: a semantic perspective. In *Handbook of Modal Logic*. Studies in Logic and Practical Reasoning, Vol. 3. Elsevier, 1–84.

[20] Patrick Blackburn, Johan FAK van Benthem, and Frank Wolter. 2006. *Handbook of Modal Logic*. Vol. 3. Elsevier.

[21] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal verification of a C compiler front-end. In *FM 2006: Formal Methods*. Springer, 460–475.

[22] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. 2003. The Zettabyte File System. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies (USENIX FAST)*.

[23] Stephen Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR*. http://link.springer.com/chapter/10.1007/978-3-540-28644-8_2

[24] James Brotherston and Jules Villard. 2014. Parametric completeness for separation theories. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 453–464.

[25] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *Proceedings of the 22Nd Annual IEEE Symposium on Logic in Computer Science (LICS '07)*. IEEE Computer Society, Washington, DC, USA, 366–378. https://doi.org/10.1109/LICS.2007.30

[26] Pierre Castéran and Yves Bertot. 2004. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* Springer Verlag. 470 pages.

[27] Tej Chajed. 2022. *Verifying a concurrent, crash-safe file system with sequential reasoning*. Ph.D. Dissertation. Machetutes Institute of Technology, Cambridge, MA. Available at https://dspace.mit.edu/handle/1721.1/144578.

[28] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 243–258. https://doi.org/10.1145/3341301.3359632

[29] Kaustuv Chaudhuri, Joëlle Despeyroux, Carlos Olarte, and Elaine Pimentel. 2019. Hybrid linear logic, revisited. *Mathematical Structures in Computer Science* 29, 8 (2019), 1151–1176.

[30] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 431–447. https://doi.org/10.1145/2908080.2908101

[31] Adam Chlipala. 2013. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 391–402. https://doi.org/10.1145/2500365.2500592

[32] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable address spaces using RCU balanced trees. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*. 199–210. https://doi.org/10.1145/2150976.2150998

[33] Ernie Coehn, Michal Moskal, Wolfram Shulte, and Stephan Tobies. 2010. *Local Verification of Global Invariants in Concurrent Programs.* Technical Report MSR-TR-2010-9. Microsoft Research.

[34] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

[35] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs) (Lecture Notes in Computer Science)*, Vol. 5674. Springer, 23–42. http://research.microsoft.com/apps/pubs/default.aspx?id=117859

[36] Ernie Cohen, Wolfgang J. Paul, and Sabine Schmaltz. 2013. Theory of Multi Core Hypervisor Verification. In *SOFSEM 2013: Theory and Practice of Computer Science, 39th International Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 26-31, 2013. Proceedings*. 1–27. https://doi.org/10.1007/978-3-642-35843-2_1

[37] Ted Cooper and Jonathan Walpole. 2015. Relativistic Programming in Haskell Using Types to Enforce a Critical Section Discipline. http://web.cecs.pdx.edu/~walpole/papers/haskell2015.pdf

[38] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. 2009. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering-Companion Volume, 2009. ICSE-Companion 2009.* IEEE, 429–430.

[39] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. 2005. On the verification of memory management mechanisms. In *Correct Hardware Design and Verification Methods*. Springer, 301–316.

[40] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL, Article 34 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371102

[41] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 34:1–34:29. https://hal.inria.fr/hal-02351793/

[42] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: strong and compositional library specifications in relaxed memory separation logic. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 792–808.

[43] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.

[44] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 337–340. http://www.audentia-gestion.fr/MICROSOFT/z3.pdf

[45] Paulo Emílio de Vilhena and François Pottier. 2023. Verifying an Effect-Handler-Based Define-By-Run Reverse-Mode AD Library. *Logical Methods in Computer Science* Volume 19, Issue 4, Article 5 (Oct 2023). https://doi.org/10.46298/lmcs-19(4:5)2023

[46] Peter J. Denning. 1970. Virtual Memory. *ACM Comput. Surv.* 2, 3 (Sept. 1970), 153–189. https://doi.org/10.1145/356571.356573

[47] Mathieu Desnoyers, Paul E. McKenney, and Michel R. Dagenais. 2013. Multi-core Systems Modeling for Formal Verification of Parallel Algorithms. *SIGOPS Oper. Syst. Rev.* 47, 2 (July 2013), 51–65. https://doi.org/10.1145/2506164.2506174

[48] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, and Jonathan Walpole. 2009. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* (2009). /static/publications/desnoyers-ieee-urcu-submitted.pdf

[49] Joëlle Despeyroux and Kaustuv Chaudhuri. 2014. A hybrid linear logic for constrained transition systems. In *Post-Proceedings of the 9th Intl. Conference on Types for Proofs and Programs (TYPES 2013)*, Vol. 26. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 150–168.

[50] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 287–300. https://doi.org/10.1145/2429069.2429104

[51] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *Principles of Programming Languages (POPL)*. 287–300. http://cs.au.dk/~birke/papers/views.pdf

[52] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. 287–300. https://doi.org/10.1145/2429069.2429104

[53] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP*.

[54] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. 2010. *Concurrent Abstract Predicates*. Technical Report. University of Cambridge, Computer Laboratory. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-777.pdf

[55] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *ESOP*.

[56] Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract*

Interpretation - Volume 9583 (VMCAI 2016). Springer-Verlag, Berlin, Heidelberg, 413–430. https://doi.org/10.1007/978-3-662-49122-5_20

[57] Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 448–475. https://doi.org/10.1007/978-3-662-54434-1_17

[58] E Allen Emerson and Edmund M Clarke. 1982. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer programming* 2, 3 (1982), 241–266.

[59] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. 2006. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*. ACM, New York, NY, USA, 177–190. https://doi.org/10.1145/1217935.1217953

[60] Xinyu Feng. 2009. Local Rely-guarantee Reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 315–327. https://doi.org/10.1145/1480881.1480922

[61] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *ESOP*.

[62] Michael J. Fischer and Richard E. Ladner. 1977. Propositional Modal Logic of Programs. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing (STOC '77)*. ACM, New York, NY, USA, 286–294. https://doi.org/10.1145/800105.803418

[63] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–402.

[64] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 4 (2014), 1–44.

[65] George Gargov and Valentin Goranko. 1993. Modal logic with names. *Journal of Philosophical Logic* 22, 6 (1993), 607–636.

[66] Valentin Goranko. 1996. Hierarchies of modal and temporal logics with reference pointers. *Journal of Logic, Language and Information* 5, 1 (1996), 1–24.

[67] Colin S Gordon. 2019. Modal assertions for actor correctness. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 11–20.

[68] Colin S. Gordon, Michael D. Ernst, Dan Grossman, and Matthew J. Parkinson. 2017. Verifying Invariants of Lock-free Data Structures with Rely-Guarantee and Refinement Types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, 3 (May 2017). https://doi.org/10.1145/3064850 URL: http://doi.acm.org/10.1145/3064850.

[69] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *Proceedings of the 2012 ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. Tucson, AZ, USA. https://doi.org/10.1145/2384616.2384619
URL: http://dl.acm.org/citation.cfm?id=2384619.

[70] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems (ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 249–269. https://doi.org/10.1007/978-3-642-37036-6_15

[71] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. 2011. CertiKOS: A Certified Kernel for Secure Cloud Computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys '11)*. ACM, New York, NY, USA, Article 3, 5 pages. https://doi.org/10.1145/2103799.2103803

[72] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

[73] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels.. In *OSDI*. 653–669.

[74] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 646–661. https://doi.org/10.1145/3192366.3192381

[75] Joseph Y Halpern. 2017. *Reasoning about uncertainty*. MIT press.

[76] Joseph Y Halpern and Ronald Fagin. 1989. Modelling knowledge and action in distributed systems. *Distributed computing* 3 (1989), 159–177.

[77] Joseph Y Halpern and Yoram Moses. 1985. A guide to the modal logics of knowledge and belief: Preliminary draft. In *Proceedings of the 9th international joint conference on Artificial intelligence-Volume 1*. Morgan Kaufmann Publishers Inc., 480–490.

[78] Joseph Y Halpern and Yoram Moses. 1990. Knowledge and common knowledge in a distributed environment. *Journal of the ACM (JACM)* 37, 3 (1990), 549–587.

[79] David Harel. 1979. First-order dynamic logic.

[80] Mark Hillebrand. 2005. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. Ph.D. Dissertation. PhD thesis, Saarland University, Computer Science Dept.

[81] Jaakko Hintikka. 1962. *Knowledge and belief*. Cornell University Press.

[82] Dave Hitz, James Lau, and Michael A Malcolm. 1994. File System Design for an NFS File Server Appliance.. In *USENIX Winter*, Vol. 94.

[83] Philip W. Howard and Jonathan Walpole. 2011. A Relativistic Enhancement to Software Transactional Memory. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism (HotPar'11)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=2001252.2001267

[84] George Edward Hughes and Max J Cresswell. 1996. *A new introduction to modal logic*.

[85] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. 2007. Sealing OS Processes to Improve Dependability and Safety. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, New York, NY, USA, 341–354. https://doi.org/10.1145/1272996.1273032

[86] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (April 2007), 37–49. https://doi.org/10.1145/1243418.1243424

[87] C. B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (Oct. 1983), 596–619. https://doi.org/10.1145/69575.69577

[88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. 1988. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems (TOCS)* 6, 1 (1988), 109–133.

[89] Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. 2023. Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 251 (Oct. 2023), 29 pages. https://doi.org/10.1145/3622827

[90] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.

[91] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants As an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

[92] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants As an Orthogonal Basis for Concurrent Reasoning. In *POPL*.

[93] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Trans. Comput. Syst.* 32, 1, Article 2 (Feb. 2014), 70 pages. https://doi.org/10.1145/2560537

[94] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[95] Michalis Kokologiannakis and Konstantinos Sagonas. 2017. Stateless Model Checking of the Linux Kernel's Hierarchical Read-copy-update (Tree RCU). In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (SPIN 2017)*. ACM, New York, NY, USA, 172–181. https://doi.org/10.1145/3092282.3092287

[96] Rafal Kolanski. 2008. A Logic for Virtual Memory. In *Systems Software Verification*, Ralf Huuck, Gerwin Klein, Bastian Schlich (Ed.). Sydney, Australia, 61–77.

[97] Rafal Kolanski and Gerwin Klein. 2008. Mapped Separation Logic. In *Verified Software: Theories, Tools and Experiments*, Natarajan Shankar, Jim Woodcock (Ed.). Springer, Toronto, Canada, 15–29.

[98] Rafal Kolanski and Gerwin Klein. 2009. Types, Maps and Separation Logic. In *International Conference on Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (Ed.). Springer, Munich, Germany, 276–292.

[99] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–30.

[100] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. In *European Symposium on Programming*. Springer, 696–723.

[101] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 205–217. https://doi.org/10.1145/3009837.3009855

[102] Saul A. Kripke. 1963. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica* 16, 1963 (1963), 83–94.

[103] H. T. Kung and Philip L. Lehman. 1980. Concurrent Manipulation of Binary Search Trees. *ACM Trans. Database Syst.* 5, 3 (Sept. 1980), 354–382. https://doi.org/10.1145/320613.320619

[104] Ismail Kuru and Colin S. Gordon. 2018. Safe Deferred Memory Reclamation with Types. *CoRR* abs/1811.11853 (2018). arXiv:1811.11853 http://arxiv.org/abs/1811.11853

[105] Ismail Kuru and Colin S. Gordon. 2019. Safe Deferred Memory Reclamation with Types. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science)*, Luís Caires (Ed.), Vol. 11423. Springer, 88–116. https://doi.org/10.1007/978-3-030-17184-1_4

[106] Ismail Kuru and Colin S. Gordon. 2024. Modal Abstractions for Virtualizing Memory Addresses. arXiv:cs.PL/2307.14471 https://arxiv.org/abs/2307.14471

[107] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++ 11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 618–632.

[108] Leslie Lamport. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 872–923.

[109] Leslie Lamport. 2002. Specifying systems: the TLA+ language and tools for hardware and software engineers. (2002).

[110] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.

[111] Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning* 41, 1 (2008), 1–31.

[112] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 234–251.

[113] Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective auxiliary state for coarse-grained concurrency. In *Principles of Programming Languages (POPL)*. 561–574. http://software.imdea.org/~aleks/papers/concur/scsl4.pdf

[114] Lihao Liang, Paul E. McKenney, Daniel Kroening, and Tom Melham. 2016. Verification of the Tree-Based Hierarchical Read-Copy Update in the Linux Kernel. *CoRR* abs/1610.03052 (2016). http://arxiv.org/abs/1610.03052

[115] J. Liedtke. 1995. On Micro-kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 237–250. https://doi.org/10.1145/224056.224075

[116] Jochen Liedtke. 1996. Toward Real Microkernels. *Commun. ACM* 39, 9 (Sept. 1996), 70–77. https://doi.org/10.1145/234215.234473

[117] John Lions. 1996. *Lions' commentary on UNIX 6th edition with source code*. Peer-to-Peer Communications, Inc.

[118] M. U. Mandrykin and A. V. Khoroshilov. 2016. Towards Deductive Verification of C Programs with Shared Data. *Program. Comput. Softw.* 42, 5 (Sept. 2016), 324–332. https://doi.org/10.1134/S0361768816050054

[119] Richard McDougall and Jim Mauro. 2006. *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. Pearson Education.

[120] Paul E. Mckenney. 2004. *Exploiting Deferred Destruction: An Analysis of Read-copy-update Techniques in Operating System Kernels*. Ph.D. Dissertation. Oregon Health & Science University. AAI3139819.

[121] Paul E. McKenney. 2014. N4037: Non-Transactional Implementation of Atomic Tree Move. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4037.pdf

[122] Paul E. McKenney. 2015. Some Examples of Kernel-Hacker Informal Correctness Reasoning *(Technical Report paulmck.2015.06.17a)*. http://www2.rdrop.com/users/paulmck/techreports/IntroRCU.2015.06.17a.pdf

[123] Paul E. Mckenney. 2017. A Tour Through RCU's Requirements. https://www.kernel.org/doc/Documentation/RCU/Design/Requirements/Requirements.html

[124] Paul E. Mckenney, Jonathan Appavoo, Andi Kleen, O. Krieger, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2001. Read-Copy Update. In *In Ottawa Linux Symposium*. 338–367.

[125] Roland Meyer and Sebastian Wolff. 2019. Decoupling lock-free data structures from memory reclamation for static analysis. *PACMPL* 3, POPL (2019), 58:1–58:31. https://dl.acm.org/citation.cfm?id=3290371

[126] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (June 2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

[127] Rasmus Ejlers Møgelberg. 2014. A type theory for productive coprogramming via guarded recursion. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, 71.

[128] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583 (VMCAI 2016)*. Springer-Verlag New York, Inc., New York, NY, USA, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

[129] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating state transition systems for fine-grained concurrent resources. In *European Symposium on Programming*. Springer, 290–310.

[130] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. 2007. Using XCAP to certify realistic systems code: Machine context management. In *Theorem Proving in Higher Order Logics*. Springer, 189–206.

[131] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

[132] Peter O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (jan 2019), 86–95. https://doi.org/10.1145/3211968

[133] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. 2004. Separation and Information Hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, New York, NY, USA, 268–280. https://doi.org/10.1145/964001.964024

[134] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.

[135] Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Informatica* 6, 4 (1976), 319–340. https://doi.org/10.1007/BF00268134

[136] Lai Jiangshan Paul E. McKenney, Mathieu Desnoyers and Josh Triplett. 2016. The RCU-barrier menagerie. https://lwn.net/Articles/573497/

[137] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*. ieee, 46–57.

[138] Vaughan R Pratt. 1976. Semantical consideration on floyo-hoare logic. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*. IEEE, 109–121.

[139] Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *ESOP*.

[140] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, Washington, DC, USA, 55–74. http://dl.acm.org/citation.cfm?id=645683.664578

[141] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage* 9, 3, Article 9 (Aug. 2013), 32 pages. https://doi.org/10.1145/2501620.2501623

[142] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. https://doi.org/10.1145/146941.146943

[143] Davide Sangiorgi. 2009. On the Origins of Bisimulation and Coinduction. *ACM Trans. Program. Lang. Syst.* 31, 4, Article 15 (May 2009), 41 pages. https://doi.org/10.1145/1516507.1516510

[144] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized Verification of Fine-grained Concurrent Programs. In *PLDI*.

[145] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 471–482. https://doi.org/10.1145/2491956.2462183

[146] Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. 2015. A separation Logic for Fictional Sequential Consistency. In *European Symposium on Programming*. Springer, 736–761.

[147] Artem Starostin. 2010. *Formal verification of demand paging*. Ph.D. Dissertation. PhD thesis, Saarland University, Computer Science Dept.

[148] Robert E Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE transactions on software engineering* 1 (1986), 157–171.

[149] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 8410. Springer, 149–168. http://cs.au.dk/~birke/papers/icap-conf.pdf

[150] Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *European Symposium on Programming*. Springer, 149–168.

[151] Hira Taqdees Syeda and Gerwin Klein. 2018. Program verification in the presence of cached address translation. In *Interactive Theorem Proving: 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings 9*. Springer, 542–559.

[152] Hira Taqdees Syeda and Gerwin Klein. 2020. Formal reasoning under cached address translation. *Journal of Automated Reasoning* 64, 5 (2020), 911–945.

[153] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying Read-copy-update in a Logic for Weak Memory. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 110–120. https://doi.org/10.1145/2737924.2737992

[154] Josep Tassarotti Tej Chajed and contributors. 2023. Post-crash modality in Perennial's Coq Mechanization. https://github.com/mit-pdos/perennial/blob/master/src/goose_lang/crash_modality.v

[155] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. 2011. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=2002181.2002192

[156] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-style Reasoning in a Logic for Higher-order Concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 377–390. https://doi.org/10.1145/2500365.2500600

[157] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 691–707. https://doi.org/10.1145/2660193.2660243

[158] Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR (Lecture Notes in Computer Science)*, Luís Caires and Vasco T. Vasconcelos (Eds.), Vol. 4703. Springer Berlin Heidelberg, Berlin, Heidelberg. http://www.springerlink.com/content/m20j47mp273414gx/

[159] Alexander Vaynberg and Zhong Shao. 2012. Compositional verification of a baby virtual memory manager. In *Certified Programs and Proofs*. Springer, 143–159.

[160] Simon Friis Vindum, Aïna Linn Georges, and Lars Birkedal. 2025. The Nextgen Modality: A Modality for Non-Frame-Preserving Updates in Separation Logic. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '25)*. Association for Computing Machinery, New York, NY, USA, 83–97. https://doi.org/10.1145/3703595.3705876

[161] Michael von Tessin. 2013. *The clustered multikernel: An approach to formal verification of multiprocessor operating-system kernels*. Ph.D. Dissertation. PhD thesis, School of Computer Science and Engineering, UNSW, Sydney, Australia, Sydney, Australia.

[162] Andrew Wagner, Zachary Eisbach, and Amal Ahmed. 2024. Realistic Realizability: Specifying ABIs You Can Count On. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 315 (Oct. 2024), 30 pages. https://doi.org/10.1145/3689755

[163] Jean Yang and Chris Hawblitzel. 2010. Safe to the Last Instruction: Automated Verification of a Type-safe Operating System. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 99–110. https://doi.org/10.1145/1806596.1806610

APPENDIX A

ASSEMBLY IMPLEMENTATION OF VIRTUAL MEMORY MANAGEMENT

A.1 Assembly Implementation of PTE Library

```
pte.o: file format elf64-x86-64



Disassembly of section .text:

0000000000000000 <pte_init>:

#include "kalloc.h"
#include "lib.h"
#include "x86.h"

0000000000000070 <pte_initialize>:
}

void pte_initialize(pte_t *entry) {
  70: 55  push %rbp
  71: 48 89 e5  mov %rsp,%rbp
  74: 48 83 ec 10  sub $0x10,%rsp
  78: 48 89 7d f8  mov %rdi,-0x8(%rbp)

    // Allocate a full page for 512 8-byte entries
    pte_t *local = kalloc();
```

```
23   7c: e8 00 00 00 00  callq 81 <pte_initialize+0x11> 7d: R_X86_64_PLT32 kalloc-0x4
24   81: 48 89 45 f0  mov %rax,-0x10(%rbp)
25
26     // Clear the entire table. Ensures present (and reserved, etc.) bits are 0
27     //memset(local, 0, PAGE_SIZE);
28     entry->pfn = PTE_ADDR_TO_PFN((uintptr_t) local);
29   85: 48 8b 45 f0  mov -0x10(%rbp),%rax
30   89: 48 c1 e8 0c  shr $0xc,%rax
31   8d: 48 8b 7d f8  mov -0x8(%rbp),%rdi
32   91: 48 8b 0f  mov (%rdi),%rcx
33   94: 48 ba ff ff ff ff 00 00 00  movabs $0xffffffff,%rdx
34   9e: 48 21 d0  and %rdx,%rax
35   a1: 48 c1 e0 0c  shl $0xc,%rax
36   a5: 48 ba ff 0f 00 00 00 00 f0 ff  movabs $0xfff0000000000fff,%rdx
37   af: 48 21 d1  and %rdx,%rcx
38   b2: 48 09 c1  or %rax,%rcx
39   b5: 48 89 0f  mov %rcx,(%rdi)
40
41     return;
42   b8: 48 83 c4 10  add $0x10,%rsp
43   bc: 5d  pop %rbp
44   bd: c3  retq
45   be: 66 90  xchg %ax,%ax
46
47 00000000000000c0 <pte_uninit>:
48 }
49
50 00000000000000e0 <pte_get_next_table_succ>:
51  * @param entry - the entry from which to get the next level of tables
52  * @param alloc - if non-zero, if the next table is not present, allocates it
53  * @return - pointer to the next page table level or, if not present and alloc
54  * is not set or memory is not sufficient, NULL
55  */
56  pte_t *pte_get_next_table_succ(pte_t *entry) {
```

```
57   e0: 55  push %rbp
58   e1: 48 89 e5  mov %rsp,%rbp
59   e4: 48 83 ec 20  sub $0x20,%rsp
60   e8: 48 89 7d f8  mov %rdi,-0x8(%rbp)
61      pte_t *next;
62
63
64      // If not already present, try to allocate
65      if (!entry->present) {
66   ec: 48 8b 7d f8  mov -0x8(%rbp),%rdi
67   f0: 48 8b 3f  mov (%rdi),%rdi
68   f3: 48 83 e7 01  and $0x1,%rdi
69   f7: 40 88 f8  mov %dil,%al
70   fa: 3c 00  cmp $0x0,%al
71   fc: 0f 85 3f 00 00 00  jne 141 <pte_get_next_table_succ+0x61>
72          // If it shouldn't be or cannot be allocated, indicate failure
73          pte_initialize(entry);
74  102: 48 8b 7d f8  mov -0x8(%rbp),%rdi
75  106: e8 65 ff ff ff  callq 70 <pte_initialize>
76
77          // Update the entry to indicate that a new table has now been allocated.
78          // Make it writable and user-accesible - if necessary, the lowest level
79          // of page table can fix these access rights. Indicate the *physical*
80          // address of the newly allocated structure in pfn.
81          entry->writable = 1;
82  10b: 48 8b 7d f8  mov -0x8(%rbp),%rdi
83  10f: 48 8b 07  mov (%rdi),%rax
84  112: 48 83 e0 fd  and $0xfffffffffffffd,%rax
85  116: 48 83 c8 02  or $0x2,%rax
86  11a: 48 89 07  mov %rax,(%rdi)
87          entry->user_acc = 1;
88  11d: 48 8b 45 f8  mov -0x8(%rbp),%rax
89  121: 48 8b 38  mov (%rax),%rdi
90  124: 48 83 e7 fb  and $0xfffffffffffffb,%rdi
```

```
128: 48 83 cf 04  or $0x4,%rdi
12c: 48 89 38  mov %rdi,(%rax)
        // entry->pfn = PTE_ADDR_TO_PFN((uintptr_t) next);
        entry->present = 1;
12f: 48 8b 45 f8  mov -0x8(%rbp),%rax
133: 48 8b 38  mov (%rax),%rdi
136: 48 83 e7 fe  and $0xfffffffffffffe,%rdi
13a: 48 83 cf 01  or $0x1,%rdi
13e: 48 89 38  mov %rdi,(%rax)
    }
    uintptr_t next_phys_addr = PTE_PFN_TO_ADDR(entry->pfn);
141: 48 8b 45 f8  mov -0x8(%rbp),%rax
145: 48 8b 00  mov (%rax),%rax
148: 48 c1 e8 0c  shr $0xc,%rax
14c: 48 b9 ff ff ff ff ff 00 00 00  movabs $0xffffffffff,%rcx
156: 48 21 c8  and %rcx,%rax
159: 48 c1 e0 0c  shl $0xc,%rax
15d: 48 89 45 e8  mov %rax,-0x18(%rbp)
    uintptr_t next_virt_addr = (uintptr_t) P2V(next_phys_addr);
161: 48 8b 45 e8  mov -0x18(%rbp),%rax
165: 48 b9 00 00 00 00 00 00 00 00  movabs $0x0,%rcx 167: R_X86_64_64 KERN_BASE
16f: 48 01 c8  add %rcx,%rax
172: 48 89 45 e0  mov %rax,-0x20(%rbp)
    next = (pte_t *) next_virt_addr;
176: 48 8b 45 e0  mov -0x20(%rbp),%rax
17a: 48 89 45 f0  mov %rax,-0x10(%rbp)

    return next;
17e: 48 8b 45 f0  mov -0x10(%rbp),%rax
182: 48 83 c4 20  add $0x20,%rsp
186: 5d  pop %rbp
187: c3  retq
188: 0f 1f 84 00 00 00 00 00  nopl 0x0(%rax,%rax,1)

```

```
0000000000000190 <walkpgdir>:
    }

    return next;
}


00000000000003e0 <walkpgdir_succ>:
pte_t *walkpgdir_succ(pte_t *pml4, const void *va) {
 3e0: 55  push %rbp
 3e1: 48 89 e5  mov %rsp,%rbp
 3e4: 48 83 ec 40  sub $0x40,%rsp
 3e8: 48 89 7d f8  mov %rdi,-0x8(%rbp)
 3ec: 48 89 75 f0  mov %rsi,-0x10(%rbp)
    pte_t *pml4_entry = &pml4[PML4EX(va)];
 3f0: 48 8b 75 f8  mov -0x8(%rbp),%rsi
 3f4: 48 8b 7d f0  mov -0x10(%rbp),%rdi
 3f8: 48 c1 ef 27  shr $0x27,%rdi
 3fc: 48 81 e7 ff 01 00 00  and $0x1ff,%rdi
 403: 48 c1 e7 03  shl $0x3,%rdi
 407: 48 01 fe  add %rdi,%rsi
 40a: 48 89 75 e8  mov %rsi,-0x18(%rbp)

    pte_t *pdp = pte_get_next_table_succ(pml4_entry);
 40e: 48 8b 7d e8  mov -0x18(%rbp),%rdi
 412: e8 a9 fc ff ff  callq c0 <pte_get_next_table_succ>
 417: 48 89 45 e0  mov %rax,-0x20(%rbp)

    pte_t *pdp_entry = &pdp[PDPEX(va)];
 41b: 48 8b 45 e0  mov -0x20(%rbp),%rax
 41f: 48 8b 75 f0  mov -0x10(%rbp),%rsi
 423: 48 c1 ee 1e  shr $0x1e,%rsi
 427: 48 81 e6 ff 01 00 00  and $0x1ff,%rsi
 42e: 48 c1 e6 03  shl $0x3,%rsi
```

```
159  432: 48 01 f0  add %rsi,%rax
160  435: 48 89 45 d8  mov %rax,−0x28(%rbp)
161
162      pte_t *pd = pte_get_next_table_succ(pdp_entry);
163  439: 48 8b 7d d8  mov −0x28(%rbp),%rdi
164  43d: e8 7e fc ff ff  callq c0 <pte_get_next_table_succ>
165  442: 48 89 45 d0  mov %rax,−0x30(%rbp)
166
167      pte_t *pd_entry = &pd[PDEX(va)];
168  446: 48 8b 45 d0  mov −0x30(%rbp),%rax
169  44a: 48 8b 75 f0  mov −0x10(%rbp),%rsi
170  44e: 48 c1 ee 15  shr $0x15,%rsi
171  452: 48 81 e6 ff 01 00 00  and $0x1ff,%rsi
172  459: 48 c1 e6 03  shl $0x3,%rsi
173  45d: 48 01 f0  add %rsi,%rax
174  460: 48 89 45 c8  mov %rax,−0x38(%rbp)
175
176      pte_t *pt = pte_get_next_table_succ(pd_entry);
177  464: 48 8b 7d c8  mov −0x38(%rbp),%rdi
178  468: e8 53 fc ff ff  callq c0 <pte_get_next_table_succ>
179  46d: 48 89 45 c0  mov %rax,−0x40(%rbp)
180
181      return &pt[PTEX(va)];
182  471: 48 8b 45 c0  mov −0x40(%rbp),%rax
183  475: 48 8b 75 f0  mov −0x10(%rbp),%rsi
184  479: 48 c1 ee 0c  shr $0xc,%rsi
185  47d: 48 81 e6 ff 01 00 00  and $0x1ff,%rsi
186  484: 48 c1 e6 03  shl $0x3,%rsi
187  488: 48 01 f0  add %rsi,%rax
188  48b: 48 83 c4 40  add $0x40,%rsp
189  48f: 5d  pop %rbp
190  490: c3  retq
191  491: 66 2e 0f 1f 84 00 00 00 00 00  nopw %cs:0x0(%rax,%rax,1)
192  49b: 0f 1f 44 00 00  nopl 0x0(%rax,%rax,1)
```

```
193
194 }
```

A.2 x86 Instructions for Mapping a Page

```
1  0000000000003f0 <vaspace_mapapage>:
2
3  void vaspace_mapapage(
4      pte_t *pml4,
5      void *va,
6      uintptr_t pa
7  ) {
8        3f0: 55  push %rbp
9        3f1: 48 89 e5  mov %rsp,%rbp
10       3f4: 48 83 ec 30  sub $0x30,%rsp
11       3f8: 48 89 7d f8  mov %rdi,-0x8(%rbp)
12       3fc: 48 89 75 f0  mov %rsi,-0x10(%rbp)
13       400: 48 89 55 e8  mov %rdx,-0x18(%rbp)
14     char *a;
15     pte_t *pte;
16
17     a = (char * ) PGROUNDUP((uintptr_t)va);
18       404: 48 8b 55 f0  mov -0x10(%rbp),%rdx
19       408: 48 81 c2 00 10 00 00  add $0x1000,%rdx
20       40f: 48 83 ea 01  sub $0x1,%rdx
21       413: 48 81 e2 00 f0 ff ff  and $0xfffffffffff000,%rdx
22       41a: 48 89 55 e0  mov %rdx,-0x20(%rbp)
23
24     pte = walkpgdir_succ(pml4, a);
25       41e: 48 8b 7d f8  mov -0x8(%rbp),%rdi
26       422: 48 8b 75 e0  mov -0x20(%rbp),%rsi
27       426: b0 00  mov $0x0,%al
28       428: e8 00 00 00 00  callq 42d <vaspace_mapapage+0x3d> 429: R_X86_64_PLT32 walkpgdir_succ-0x4
```

```
      42d: 48 63 d0  movslq %eax,%rdx
      430: 48 89 55 d8  mov %rdx,−0x28(%rbp)
    pte−>pfn = PTE_ADDR_TO_PFN(pa);
      434: 48 8b 55 e8  mov −0x18(%rbp),%rdx
      438: 48 c1 ea 0c  shr $0xc,%rdx
      43c: 48 8b 75 d8  mov −0x28(%rbp),%rsi
      440: 48 8b 3e  mov (%rsi),%rdi
      443: 48 b9 ff ff ff ff ff 00 00 00  movabs $0xffffffffff,%rcx
      44d: 48 21 ca  and %rcx,%rdx
      450: 48 c1 e2 0c  shl $0xc,%rdx
      454: 48 b9 ff 0f 00 00 00 00 f0 ff  movabs $0xfff0000000000fff,%rcx
      45e: 48 21 cf  and %rcx,%rdi
      461: 48 09 d7  or %rdx,%rdi
      464: 48 89 3e  mov %rdi,(%rsi)

    return;
      467: 48 83 c4 30  add $0x30,%rsp
      46b: 5d  pop %rbp
      46c: c3  retq
      46d: 0f 1f 00  nopl (%rax)

00000000000004a0 <kvm_init>:

success:
    return error;
}
```

APPENDIX B

COMPLETE SOUNDNESS PROOF OF ATOMS AND STRUCTURAL PROGRAM STATEMENTS

B.1 Complete Constructions for **Views**

To prove soundness we use the Views Framework[52]. The Views Framework takes a set of parameters satisfying some properties, and produces a soundness proof for a static reasoning system for a larger programming language. Among other parameters, the most notable are the choice of machine state, semantics for *atomic* actions (e.g., field writes, or WriteBegin), and proofs that the reasoning (in our case, type rules) for the atomic actions are sound (in a way chosen by the framework). The other critical pieces are a choice for a partial *view* of machine states — usually an extended machine state with meta-information — and a relation constraining how other parts of the program can interfere with a view (e.g., modifying a value in the heap, but not changing its type). Our type system will be related to the views by giving a denotation of type environments in terms of views, and then proving that for each atomic action shown in 2.1 in Section 2 and type rule in Figures 3.3 Section 3.2 and F.1 Appendix F, 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

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

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

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

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

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

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

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

that projects instrumented states to the common components with MState.

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

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

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

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

Figure B.1 Type Environments

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 B.1. 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 B.2 An important property of composition is that it preserves validity of logical states:

Lemma 6 (Well Formed Composition) *Any successful composition of two well-formed logical*

$$\bullet = (\bullet_\sigma, \bullet_O, \cup, \cup) \quad O_1 \bullet_O O_2(loc) \stackrel{\text{def}}{=} O_1(loc) \cup O_2(loc) \quad (s_1 \bullet_s s_2) \stackrel{\text{def}}{=} s_1 \cup s_2 \ \texttt{when}\ dom(s_1) \cap dom(s_2) = \emptyset$$

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

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

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

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

Figure B.2 Composition($\bullet$) and Thread Interference Relation($\mathcal{R}_0$)

states is well-formed:

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

Proof: By assumption, we know that Wellformed(x) and Wellformed(y) hold. We need to show that composition of two well-formed states preserves well-formedness which is to show that for all z such that $x \bullet y = z$, Wellformed(z) holds. Both x and y have 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. $\square$ We define separation on elements of type contexts

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

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

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

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

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

Thread interference relation

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

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

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

where $\mathcal{R}$ is transitive-reflexive closure of $\mathcal{R}_0$ shown at Figure B.2. $\mathcal{R}_0$ (and therefore $\mathcal{R}$) also "preserves" validity:

Lemma 7 (Valid $\mathcal{R}_0$ Interference) *For any m and m', if* $\mathsf{WellFormed}(m)$ *and $m\mathcal{R}_0 m'$, then*

$\mathsf{WellFormed}(m')$.

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

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

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

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

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

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

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

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

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

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

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

is stable, and must show that

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

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

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

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

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

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

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

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

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

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

prove this case.

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

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

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

prove this case.

□

Lemma 9 (Stable Environment Denotation-R) *For any* closed *environment* Γ *(i.e.,* $\forall x \in \mathsf{dom}(\Gamma).\, , \mathsf{FV}(\Gamma(x)) \subseteq \mathsf{dom}(\Gamma)$*):*

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

$$[\![\mathsf{if}\ (x.f == y)\ C_1\ C_2]\!]_{tid} \stackrel{\mathsf{def}}{=} z = x.f; ((\mathsf{assume}(z = y); C_1) + (\mathsf{assume}(z \neq y); C_2)) \quad [\![\mathtt{assume}(\mathcal{S})]\!](s) \stackrel{\mathsf{def}}{=} \begin{cases} \{s\} & \text{if } s \in \mathcal{S} \\ \emptyset & \text{Otherwise} \end{cases}$$

$$[\![\mathsf{while}\ (e)\ C]\!] \stackrel{\mathsf{def}}{=} (\mathsf{assume}(e); C)^*; (\mathsf{assume}(\neg e)); \frac{\{P\} \cap \{\lceil \mathcal{S} \rceil\} \sqsubseteq \{Q\}}{\{P\}\mathtt{assume}\,(b)\,\{Q\}} \quad \text{where} \quad \lceil \mathcal{S} \rceil = \{m | \lfloor m \rfloor \cap \mathcal{S} \neq \emptyset\}$$

Figure B.3 Encoding of assume(b)

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

Proof: Proof is similar to one for Lemma 8 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 3.3 Section 3.3, Figure 3.2 Section 3.1 and Figure F.1 Appendix F) 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 B.3. assume(b) is a standard construct in verification semantics[13] [128], 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[68,69].

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:$p \sqsubseteq q \stackrel{def}{=} \forall m \in \mathcal{M}. \lfloor p * \{m\} \rfloor \subseteq \lfloor q * \mathcal{R}(\{m\}) \rfloor$. 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[25,140].

Lemma 10 (Axiom of Soundness for Atoms) *For each axiom,* $\Gamma_1 \vdash_{RMO} \alpha \dashv \Gamma_2$*, we must show*

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

Proof: By case analysis on the atomic action α followed by inversion on typing derivation. All the cases proved as different lemmas in Section B.3. □

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

Lemma 11 (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 12 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 12 (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: rcultr $\rho \, \mathcal{N}$, undef, rcuFresh, unlinked and freeable. Denotations of these types include the constraint $t = l$ and other constraints specific to the type's denotation. The set of logical state defined by the denotation of the type is *subset* of intersection of the set of logical states defined by $[\![M]\!]_t \cap [\![x : T]\!]_t$ which shrinks down the logical states defined by $[\![M]\!]_t = \{((s, h, l, rt, R, B), O, U, T, F) | t = l\}$ to the set of logical states defined by denotation $[\![x : T]\!]_t$. □ Because the intersection of the environment denotation with the denotations for the different critical sections remains a valid view, the Views Framework provides most of this proof for free, given corresponding lemmas for the *atomic actions* α:

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

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

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 13 (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. □

B.2 Complete Memory Axioms

1. Ownership invariant in Figure B.4 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 B.5 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 B.6 asserts that the unique root location can only be aliased with thread local references through which the unique root location is observed as iterator.

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

Figure B.4 Ownership

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

Figure B.5 Reader-Writer-Iterator-Coexistence-Ownership

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

Figure B.6 Alias with Unique Root

4. Iterators-Free-List invariant in Figure B.7 asserts that if a heap location is observed as iterator and it is the free list then the observer thread is in the set of bounding threads.

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

Figure B.7 Iterators-Free-List

5. Unlinked-Reachability invariant in Figure B.8 asserts that if a heap node is observed as unlinked then all heap locations from which you can reach to the unlinked one are also unlinked or in the free list.

6. Free-List-Reachability invariant in Figure B.9 asserts that if a heap location is in the free list then all heap locations from which you can reach to the one in the free list are also in the free list.

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

Figure B.8 Unlinked-Reachability

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

Figure B.9 Free-List-Reachability

7. Writer-Unlink invariant in Figure B.10 asserts that the writer thread cannot observe a heap location as unlinked.

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

Figure B.10 Writer-Unlink

8. Fresh-Reachable invariant in Figure B.11 asserts that there exists no heap location that can reach to a freshly allocated heap location together with fact on nonexistence of aliases to it.

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

Figure B.11 Fresh-Reachable

9. Fresh-Writer invariant in Figure B.12 asserts that heap allocation can be done only by writer thread.

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

Figure B.12 Fresh-Writer

10. Fresh-Not-Reader invariant in Figure B.13 asserts that a heap location allocated freshly cannot be observed as unlinked or iterator.

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

Figure B.13 Fresh-Not-Reader

11. Fresh-Points-Iterator invariant in Figure B.14 states that any field of fresh allocated object can only be set to point heap node which can be observed as iterator (not unlinked or freeable). This invariant captures the fact $\mathcal{N} = \mathcal{N}'$ asserted in the type rule for fresh node linking(T-Replace).

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

Figure B.14 Fresh-Points-Iterator

12. Writer-Not-Reader invariant in Figure B.15 asserts that a writer thread identifier can not be a reader thread identifier.

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

Figure B.15 Writer-Not-Reader

13. Readers-Iterator-Only invariant in the Figure B.16 asserts that a reader threads can only make iterator observation on a heap location.

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

Figure B.16 Readers-Iterator-Only

14. Readers-In-Free-List invariant in Figure B.17 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).

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

Figure B.17 Readers-In-Free-List

15. Heap-Domain invariant in the Figure B.18 defines the domain of the heap.

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

Figure B.18 Heap-Domain

16. Unique-Root invariant in Figure B.19 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*.

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

Figure B.19 Unique-Root

17. Unique-Reachable invariant in Figure B.20 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.

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

Figure B.20 Unique-Reachable

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

B.3 Soundness Proof of Atoms

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

Lemma 14 (Unlink)

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

Proof: We assume

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

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

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

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

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

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

We split the composition in B.1 as

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

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

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

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

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

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

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

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

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

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

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

(B.16)

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

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

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

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

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

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

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

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

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

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

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

and B.26 is determined by operational semantics.

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

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

B.28 follows from B.1

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

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

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

In the rest of the proof, we prove B.24, B.25 and show the composition of $(\sigma_1', O_1', U_1', T_1', F_1')$ and $(\sigma_2', O_2', U_2', T_2', F_2')$. To prove B.24, we need to show that each of the memory axioms in Section B.2 holds for the state $(\sigma', O_1', U_1', T_1', F_1')$.

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

Case 4 - *UNQR* *B.29 and B.30 follow from framing assumption(B.3-B.5), denotations of the precondition(B.6) and B.13.* ***UNQR***

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

and

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

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

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

to show that uniqueness is preserved.

We know from operational semantics that root has not changed so

$$\sigma.rt = \sigma'.rt$$

From denotations (B.15) 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* $\in O_1'(\sigma'.h^*(\sigma.rt, \rho.f_1))$*). B.17 is determined directly by operational semantics.*

unlinked $\in O_1'(o_y)$ *follows from B.27 and B.26 which makes path $\rho.f_1.f_2$ invalid(from denotation(B.15), all heap locations reaching to $O_1'(o_r)$ from root($\sigma.rt$) are observed as* iterator *tid so this proves that* 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 B.32 and B.33*

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

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

from which B.31 follows.

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

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

Case 7 - ***IFL*** *By B.13.**WULK**, B.13.**RINFL**, B.13.**IFL**, B.27 and choice of F_1'.*

Case 8 - ***FLR*** *By choice of F_1' and B.13.*

Case 9 - ***WULK*** *By B.15, B.27 and B.28.*

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

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

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

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

Case 14 - ***RINFL*** *By B.15, B.13.**RINFL**, choice of F_1' and B.26.*

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

$$\forall_{o',f'}.\,\sigma'.h(o',f') = o_y \implies \mathit{unlinked} \in O'_1(o') \vee (\mathit{freeable} \in O'_1(o')) \tag{B.34}$$

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

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

To prove B.18 we need to show interference relation

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

which by definition means that we must show

$$\sigma_2.l \in T_2 \to (\sigma_2.h = \sigma'_2.h \wedge \sigma_2.l = \sigma'_2.l) \tag{B.35}$$

$$l \in T_2 \to F_2 = F'_2 \tag{B.36}$$

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

$$\forall tid, o.\,\mathsf{iterator}\ tid \in O_2(o) \to o \in dom(\sigma'_2.h) \tag{B.38}$$

$$O_2 = O'_2 \wedge U_2 = U'_2 \wedge T_2 = T'_2 \wedge \sigma_2.R = \sigma'_2.R \wedge \sigma_2.rt = \sigma'_2.rt \tag{B.39}$$

$$\forall x, t \in T_2.\,\sigma_2.s(x,t) = \sigma'_2.s(x,t) \tag{B.40}$$

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

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

To prove all relations (B.35-B.40) we assume B.28 which is to assume T_2 as subset of reader threads. Let σ'_2 be σ_2. O_2 need not change so we pick O'_2 as O_2. Since T_2 is subset of reader threads, we pick T_2 as T'_2. We pick F'_2 as F_2.

B.35 and B.36 follow from B.28 and choice of F'_2. B.41, B.42 and B.39 are determined by choice of

σ_2', operational semantic and choices made on maps related to the assertions.

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

To prove B.20 consider two cases: $O_1' \cap O_2' = \emptyset$ and $O_1' \cap O_2' \neq \emptyset$. The first case is trivial. The second case is where we consider

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

We also know from B.27 that

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

Both together with B.9 and B.15 proves B.20.

To show B.19 we consider two cases: $\sigma_1'.h \cap \sigma_2'.h = \emptyset$ and $\sigma_1'.h \cap \sigma_2'.h \neq \emptyset$. First is trivial. Second follows from B.24.**OW-HD** and B.25.**OW-HD**. B.21, B.22 and B.23 are trivial by choices on related maps and semantics of composition operations on them. All compositions shown let us to derive conclusion for $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$. □

Lemma 15 (Replace)

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

Proof: We assume

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

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

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

$$\mathsf{FV}(\Gamma) \cap \{p, r, n\} = \emptyset \tag{B.45}$$

$$\rho.f = \rho' \text{ and } \mathcal{N}(f) = r \tag{B.46}$$

$$\mathcal{N}' = \mathcal{N}'' \tag{B.47}$$

$$\forall_{x \in \Gamma, \mathcal{N}''', \rho'', f', y}.\, (x : \mathsf{rcultr}\ \rho''\ \mathcal{N}'''([f' \rightharpoonup y])) \implies (\neg \mathsf{MayAlias}(\rho'', \{\rho, \rho'\}) \wedge (y \neq o)) \tag{B.48}$$

We split the composition in B.43 as

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

$$(\sigma_2, O_2, U_2, T_2, F_2) = m \tag{B.50}$$

$$O_1 \bullet_O O_2 = O \tag{B.51}$$

$$\sigma_1 \bullet_s \sigma_2 = \sigma \tag{B.52}$$

$$U_1 \cup U_2 = U \tag{B.53}$$

$$T_1 \cup T_2 = T \tag{B.54}$$

$$F_1 \uplus F_2 = F \tag{B.55}$$

$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1) \tag{B.56}$$

$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{B.57}$$

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

$$(\sigma_1', O_1', U_1', T_1', F_1') \in [\![p : \mathsf{rcultr}\ \rho\ \mathcal{N}\ , n : \mathsf{rcultr}\ \rho'\ \mathcal{N}''\ , r : \mathsf{unlinked}\ , \Gamma]\!]_{M,tid} \tag{B.58}$$

$$\mathcal{N}(f) = n \tag{B.59}$$

$$(\sigma_2', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{B.60}$$

$$O_1' \bullet_O O_2' = O' \tag{B.61}$$

$$\sigma_1' \bullet_s \sigma_2' = \sigma' \tag{B.62}$$

$$U_1' \cup U_2' = U' \tag{B.63}$$

$$T_1' \cup T_2' = T' \tag{B.64}$$

$$F_1' \uplus F_2' = F' \tag{B.65}$$

$$\mathsf{WellFormed}(\sigma_1', O_1', U_1', T_1', F_1') \tag{B.66}$$

$$\mathsf{WellFormed}(\sigma_2', O_2', U_2', T_2', F_2') \tag{B.67}$$

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

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

B.59 is determined directly from operational semantics.

We know that changes in observation map are

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

and

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

B.71 follows from B.43

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

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 B.49 and choices on maps makes $(\sigma_1', O_1', U_1', T_1')$ in denotation

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

In the rest of the proof, we prove B.66, B.67 and show the composition of $(\sigma_1', O_1', U_1', T_1', F_1')$ and $(\sigma_2', O_2', U_2', T_2', F_2')$. To prove B.66, we need to show that each of the memory axioms in Section B.2 holds for the state $(\sigma', O_1', U_1', T_1', F_1')$.

Case 17 - *UNQR* *Let o_p be $\sigma.s(p, tid)$, o_r be $\sigma.s(r, tid)$ and o_n be $\sigma.s(n, tid)$. B.71 and B.73 follow from framing assumption(B.45-B.48), denotations of the precondition(B.49), B.13.**FR** and B.56.**UNQR***

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

and

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

*where o_p, o_r are $\sigma.h^*(\sigma.rt, \rho)$, $\sigma.h^*(\sigma.rt, \rho.f)$ respectively and they(heap locations in B.73 and paths in B.72) are unique(From B.56.**FR**, we assume that there exists no field alias/path alias to heap location freshly allocated o_n).*

We must prove

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

We know from operational semantics that root has not changed so

$$\sigma.rt = \sigma'.rt$$

From denotations (B.58) we know that all heap locations reached by following ρ and $\rho.f$ are observed as iteartor *tid including the final reached heap locations(*iterator *tid* $\in O'_1(\sigma'.h^*(\sigma.rt, \rho))$, iterator *tid* $\in O'_1(\sigma'.h^*(\sigma.rt, \rho.f))$ *and* iterator *tid* $\in O'_1(\sigma'.h^*(\sigma.rt, \rho.f.f_i)))$. *The preservation of uniqueness follows from B.69, B.70, B.68 and B.56.**FR**.*

from which we conclude B.75 and B.76

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

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

from which B.74 follows.

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

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

Case 20 - ***AWRT*** *Trivial.*

Case 21 - ***IFL*** *By B.56.**WULK**, B.69, B.70 choice of F'_1 and operational semantics.*

Case 22 - ***FLR*** *By choice of F'_1 and B.56.*

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

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

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

Case 26 - ***HD***

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

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

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

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

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

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

To prove B.60, we need to show interference relation

$$(\sigma, O_2, U_2, T_2, F_2)\mathcal{R}(\sigma', O_2', U_2', T_2', F_2')$$

which by definition means that we must show

$$\sigma_2.l \in T_2 \to (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{B.78}$$

$$l \in T_2 \to F_2 = F_2' \tag{B.79}$$

$$\forall tid, o.\,\mathsf{iterator}\ tid \in O_2(o) \to o \in dom(\sigma_2.h) \tag{B.80}$$

$$\forall tid, o.\,\mathsf{iterator}\ tid \in O_2(o) \to o \in dom(\sigma_2'.h) \tag{B.81}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{B.82}$$

$$\forall x, t \in T_2.\,\sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{B.83}$$

$$\forall tid, o.\,\mathsf{root}\ tid \in O(o) \to o \in dom(h) \tag{B.84}$$

$$\forall tid, o.\,\mathsf{root}\ tid \in O(o) \to o \in dom(h') \tag{B.85}$$

To prove all relations (B.78-B.83) we assume B.71 which is to assume T_2 as subset of reader threads. Let σ_2' be σ_2, F_2' be F_2. O_2 need not change so we pick O_2' as O_2. Since T_2 is subset of reader threads, we pick T_2 as T_2'. By assuming B.57 we show B.67. B.80 and B.81 follow trivially. B.83 follows from choice of σ_2', B.68 and B.71.

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

To prove B.61 consider two cases: $O_1' \cap O_2' = \emptyset$ and $O_1' \cap O_2' \neq \emptyset$. The first case is trivial. The second case is where we consider B.86 and B.87

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

From B.69 we know that

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

Both together with B.51 and B.58 proves B.61.

For case B.87

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

From B.70 we know that

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

Both together with B.51 and B.58 proves B.61.

To show B.62 we consider two cases: $\sigma_1' \cap \sigma_2' = \emptyset$ and $\sigma_1' \cap \sigma_2' \neq \emptyset$. First is trivial. Second follows from B.66.**OW-HD** and B.67.**OW-HD**. B.63, B.65 and B.64 are trivial by choices on related maps and semantics of the composition operators for these maps. All compositions shown let us to derive conclusion for $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$. □

Lemma 16 (Insert)

$$[\![p.f := n]\!](\lfloor [\![\Gamma,\ p : \mathit{rcultr}\ \rho\ \mathcal{N}\ , r : \mathit{rcultr}\ \rho_1\ \mathcal{N}_2\ , n : \mathit{rcuFresh}\ \mathcal{N}_1]\!]_{M,tid} * \{m\}\rfloor) \subseteq$$

$$\lfloor [\![\Gamma\ , p : \mathit{rcultr}\ \rho\ \mathcal{N}([f \rightharpoonup r \setminus n])\ , n : \mathit{rcultr}\ \rho_1\ \mathcal{N}_1\ , r : \mathit{rcultr}\ \rho_2\ \mathcal{N}_2]\!] * \mathcal{R}(\{m\})\rfloor$$

Proof: We assume

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

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

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

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

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

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

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

We split the composition in B.88 as

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We know that changes in observation map are

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

B.114 follows from B.88

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

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 B.94 and choices on maps makes $(\sigma_1', O_1', U_1', T_1')$ in denotation

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

In the rest of the proof, we prove B.110, B.111 and show the composition of $(\sigma_1', O_1', U_1', T_1', F_1')$ and $(\sigma_2', O_2', U_2', T_2', F_2')$. To prove B.110, we need to show that each of the memory axioms in Section B.2 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 15 and Lemma 14 with a simpler fact to prove: we assume framing conditions B.90-B.93 together with the B.101.**UNQR** and B.101.**FR** which makes B.110**UNQR** trivial.

To prove B.104, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{B.115}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{B.116}$$

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{B.117}$$

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{B.118}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{B.119}$$

$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{B.120}$$

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h) \tag{B.121}$$

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h') \tag{B.122}$$

To prove all relations (B.78-B.83) we assume B.114 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 B.102 we show B.111. B.117 and B.118 follow trivially. B.120 follows from choice of σ_2' and B.114.

B.115 and B.116 follow from B.114 and choice of F_2'. B.119, B.121 and B.122 are determined by choice of σ_2', operational semantics and choices made on maps related to the assertions.

B.105 follows from assumptions B.113, B.96 and choice of O_2' as O_2.

To show B.106 we consider two cases: $\sigma_1' \cap \sigma_2' = \emptyset$ and $\sigma_1' \cap \sigma_2' \neq \emptyset$. First is trivial. Second follows from B.110.**OW-HD** and B.111.**OW-HD**. B.107, B.109 and B.108 are trivial by choices on related maps and semantics of the composition operators for these maps. All compositions shown let us to derive conclusion for $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$. □

Lemma 17 (ReadStack)

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

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

Proof: We assume

$$(\sigma, O, U, T, F)\ \in [\![\Gamma\ , z : \_\ , x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} * \{m\} \tag{B.123}$$

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{B.124}$$

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

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

We split the composition in B.123 as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![\Gamma\ , z : \_\ , x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} \tag{B.126}$$

$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{B.127}$$

$$\sigma_1 \bullet \sigma_2 = \sigma \tag{B.128}$$

$$O_1 \bullet_O O_2 = O \tag{B.129}$$

$$U_1 \cup U_2 = U \tag{B.130}$$

$$T_1 \cup T_2 = T \tag{B.131}$$

$$F_1 \uplus F_2 = F \tag{B.132}$$

$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1) \tag{B.133}$$

$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{B.134}$$

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

$$(\sigma', O_1', U_1', T_1', F_1') \in [\![\Gamma\ , x : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}\ , z : \mathsf{rcultr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} \tag{B.135}$$

$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{B.136}$$

$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{B.137}$$

$$O_1' \bullet_O O_2' = O' \tag{B.138}$$

$$U_1' \cup U_2' = U' \tag{B.139}$$

$$T_1' \cup T_2' = T' \tag{B.140}$$

$$F_1' \uplus F_2' = F' \tag{B.141}$$

$$\mathsf{WellFormed}(\sigma_1', O_1', U_1', T_1', F_1') \tag{B.142}$$

$$\mathsf{WellFormed}(\sigma_2', O_2', U_2', T_2', F_2') \tag{B.143}$$

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

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

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

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

B.146 follows from B.123

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

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 B.126 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 B.142, B.143 and show the composition of $(\sigma_1', O_1', U_1', T_1', F_1')$ and $(\sigma_2', O_2', U_2', T_2', F_2')$. B.142 follows from B.133 trivially.

To prove B.136, we need to show interference relation

$$(\sigma, O_2, U_2, T_2, F_2)\mathcal{R}(\sigma', O_2', U_2', T_2', F_2')$$

which by definition means that we must show

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

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

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

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

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

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

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

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

To prove all relations (B.147-B.152) we assume B.146 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 B.134 we show B.143. B.149, B.150, B.153 and B.154 follow trivially. B.152 follows from choice of σ_2' and B.144(determined by operational semantics).

B.147 and B.148 follow from B.146 and choice of F_2'. B.151, B.153 and B.154 are determined by

choice of σ_2', operational semantics and choices made on maps related to the assertions.

B.138-B.141 are trivial by choices on related maps and semantics of the composition operators for these maps. B.137 follows trivially from B.128. All compositions shown let us to derive conclusion for $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$ trivial. □

Lemma 18 (ReadHeap)

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

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

Proof: We assume

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

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{B.156}$$

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

$$\mathsf{FV}(\Gamma) \cap \{z\} = \emptyset \tag{B.157}$$

$$\rho.f = \rho' \tag{B.158}$$

(B.159)

We split the composition in B.155 as

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

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

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

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

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

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

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

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

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

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

$$(\sigma', O_1', U_1', T_1', F_1) \in [\![\Gamma\ , x : \mathsf{rcultr}\ \rho\ \mathcal{N}[f \mapsto z]\ , z : \mathsf{rcultr}\ \rho'\ \mathcal{N}_\emptyset]\!]_{M,tid} \tag{B.169}$$

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

$$(\sigma', O_2', U_2', T_2', F_2) \in \mathcal{R}(\{m\}) \tag{B.171}$$

$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{B.172}$$

$$O_1' \bullet_O O_2' = O' \tag{B.173}$$

$$U_1' \cup U_2' = U' \tag{B.174}$$

$$T_1' \cup T_2' = T' \tag{B.175}$$

$$F_1' \uplus F_2' = F' \tag{B.176}$$

$$\mathsf{WellFormed}(\sigma_1', O_1', U_1', T_1', F_1') \tag{B.177}$$

$$\mathsf{WellFormed}(\sigma_2', O_2', U_2', T_2', F_2') \tag{B.178}$$

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

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

B.170 is determined directly from operational semantics.

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

$$O_1' = O_1 \tag{B.180}$$

B.181 follows from B.155

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

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 B.160 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 B.177, B.178 and show the composition of $(\sigma_1', O_1', U_1', T_1', F_1')$ and $(\sigma_2', O_2', U_2', T_2', F_2')$.

To prove B.171, we need to show that each of the memory axioms in Section B.2 holds for the state $(\sigma', O_1', U_1', T_1', F_1')$.

Case 31 - ***UNQR*** *By B.179, B.167.****UNQR*** *and* $\sigma.rt = \sigma.rt'$.

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

Case 33 - ***RWOW*** *By B.179, B.180 and B.167.****RWOW***

Case 34 - ***AWRT*** *Trivial.*

Case 35 - ***IFL*** *By B.169, B.167.****WULK****, B.180, choice of* F_1' *and operational semantics.*

Case 36 - ***FLR*** *By operational semantics(B.179), choice for* F_1' *and B.167.*

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

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

Case 39 - ***HD***

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

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

Case 42 - ***ULKR*** *Trivial.*

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

To prove B.171, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{B.182}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{B.183}$$

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{B.184}$$

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{B.185}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R_2 = \sigma_2'.R_2 \wedge \sigma_2.rt = \sigma_2'.rt \tag{B.186}$$

$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{B.187}$$

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h) \tag{B.188}$$

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h') \tag{B.189}$$

To prove all relations (B.182-B.187) we assume B.181 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 B.168 we show B.178. B.184 and B.185 follows trivially. B.187 follows from choice of σ_2' and B.179(determined by operational semantics).

B.182 and B.183 follow from B.181 and choice of F_2'.B.186, B.188 and B.189 are determined by choice of σ_2', operational semantics and choices made on maps related to the assertions.

B.173-B.176 are trivial by choices on related maps and semantics of the composition operators for these maps. B.172 follows trivially from B.162. All compositions shown let us to derive conclusion for $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$. □

Lemma 19 (WriteFreshField)

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

$$\lfloor [\![\Gamma\ , p : \mathsf{rcuFresh}\ \ \mathcal{N}(\cup_{f \rightharpoonup z})\ , x : \mathsf{rcuItr}\, \rho\, \mathcal{N}([f \rightharpoonup z])]\!] * \mathcal{R}(\{m\})\rfloor$$

Proof: We assume

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

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{B.191}$$

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

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

We split the composition in B.190 as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![\Gamma, p : \mathsf{rcuFresh}\ \mathcal{N}'_{f,\emptyset},\ x : \mathsf{rcuItr}\ \ \rho\ \ \mathcal{N}]\!]_{M,tid} \tag{B.193}$$

$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{B.194}$$

$$\sigma_1 \bullet \sigma_2 = \sigma \tag{B.195}$$

$$O_1 \bullet_O O_2 = O \tag{B.196}$$

$$U_1 \cup U_2 = U \tag{B.197}$$

$$T_1 \cup T_2 = T \tag{B.198}$$

$$F_1 \uplus F_2 = F \tag{B.199}$$

$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1) \tag{B.200}$$

$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{B.201}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

There exists no change in the observation of heap locations

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

B.214 follows from B.190

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

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 B.193 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 B.210 and B.211 and show the composition of $(\sigma_1', O_1', U_1', T_1', F_1')$ and $(\sigma_2', O_2', U_2', T_2', F_2')$. To prove B.210, we need to show that each of the memory axioms in Section B.2 holds for the state $(\sigma', O_1', U_1', T_1', F_1')$.

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

Case 45 - *OW* *By B.212,B.213 and B.200.**OW***

Case 46 - *RWOW* *By B.212, B.213 and B.200.**RWOW***

Case 47 - *AWRT* *Trivial.*

Case 48 - *IFL* *By B.200.**WULK**, B.213, choice of F_1' and operational semantics.*

Case 49 - *FLR* *By operational semantics(B.212), choice of F_1' and B.200.*

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

Case 51 - *WF* *By B.200.**WF**, B.214, B.213 and operational semantics(B.212).*

Case 52 - *FR* *By B.200.**FR**, B.214, B.213 and operational semantics(B.212).*

Case 53 - *FNR* *By B.200.**FNR**, B.214, B.213 and operational semantics(B.212).*

Case 54 - *FPI* *By B.200.**FPI**, B.193 and B.192*

Case 55 - *HD*

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

Case 57 - *RINFL* *By operational semantics(B.212 - bounding threads have not changed), choice of F_1' and B.200.*

Case 58 - *ULKR* *Trivial.*

Case 59 - *UNQRT* *By B.200.**UNQRT**, B.213 and B.212.*

To prove B.204, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{B.215}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{B.216}$$

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

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{B.218}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{B.219}$$

$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{B.220}$$

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

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h') \tag{B.222}$$

To prove all relations (B.215-B.220) we assume B.214 which is to assume T_2 as subset of reader threads and B.201. 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 B.201 we show B.211. B.217 and B.218 follows trivially. B.220 follows from choice of σ_2' and B.212(determined by operational semantics).

B.215 and B.216 follow from B.214 and choice of F_2'. B.219 are determined by operational semantics, choice of σ_2' and choices made on maps related to the assertions.

B.207-B.209 are trivial by choices on related maps and semantics of the composition operators for these maps. B.221 and B.222 follow from choice of σ_2'.

$O_1' \bullet O_2'$ follows from B.196, B.213 and choice of O_2.

We assume $\sigma_1.h \bullet \sigma_2.h$. We know from B.192 that $f \notin dom(\mathcal{N}')$. From B.202, B.210-B.211.**FNR**,

B.210-B.211.**RITR** and B.210-B.211.**WNR** we show $\sigma_1'.h \bullet \sigma_2'.h$ (with choices for other maps in the machine state we show B.172). All compositions shown let us to derive conclusion for $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$. □

Lemma 20 (Sycn)

$$[\![\mathit{SyncStart}; \mathit{SyncStop}]\!](\lfloor [\![\Gamma]\!]_{M,tid} * \{m\} \rfloor) \subseteq$$

$$\lfloor [\![\Gamma[\overline{x : \mathit{freeable}/x : \mathit{unlinked}}]]\!] * \mathcal{R}(\{m\}) \rfloor$$

Proof: We assume

$$(\sigma, O, U, T, F) \in [\![\Gamma]\!]_{M,tid} * \{m\} \tag{B.223}$$

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{B.224}$$

We split the composition in B.223 as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![\Gamma]\!]_{M,tid} \tag{B.225}$$

$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{B.226}$$

$$\sigma_1 \bullet \sigma_2 = \sigma \tag{B.227}$$

$$O_1 \bullet_O O_2 = O \tag{B.228}$$

$$U_1 \cup U_2 = U \tag{B.229}$$

$$T_1 \cup T_2 = T \tag{B.230}$$

$$F_1 \uplus F_2 = F \tag{B.231}$$

$$\mathsf{WellFormed}(\sigma, O_1, U_1, T_1, F_1) \tag{B.232}$$

$$\mathsf{WellFormed}(\sigma, O_2, U_2, T_2, F_2) \tag{B.233}$$

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

$$(\sigma', O_1', U_1', T_1', F_1') \in [\![\Gamma[\overline{x : \mathsf{freeable}/x : \mathsf{unlinked}}]]\!]_{M,tid} \tag{B.234}$$

$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{B.235}$$

$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{B.236}$$

$$O_1' \bullet_O O_2' = O' \tag{B.237}$$

$$U_1' \cup U_2' = U' \tag{B.238}$$

$$T_1' \cup T_2' = T' \tag{B.239}$$

$$\tag{B.240}$$

$$F_1' \uplus F_2' = F' \tag{B.241}$$

$$\mathsf{WellFormed}(\sigma', O_1', U_1', T_1', F_1') \tag{B.242}$$

$$\mathsf{WellFormed}(\sigma', O_2', U_2', T_2', F_2') \tag{B.243}$$

We also know from operational semantics that SyncStart changes

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

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

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

There exists no change in the observation of heap locations

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

and we pick free list to be

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

B.248 follows from B.223

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

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 B.225 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 B.242 and B.243 and show the composition of $(\sigma_1', O_1', U_1', T_1', F_1')$ and $(\sigma_2', O_2', U_2', T_2', F_2')$. To prove B.242, we need to show that each of the memory axioms in Section B.2 holds for the state $(\sigma', O_1', U_1', T_1', F_1')$ which is trivial by assuming B.232. We also know B.234(as we showed the support of state to the denotation).

To prove B.235, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{B.249}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{B.250}$$

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{B.251}$$

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{B.252}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{B.253}$$

$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{B.254}$$

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h) \tag{B.255}$$

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h') \tag{B.256}$$

To prove all relations (B.249-B.254) we assume B.248 which is to assume T_2 as subset of reader threads and B.233. 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 B.233 we show B.243. B.251 and B.252 follows trivially. B.254 follows from choice of σ_2' and B.245(determined by operational semantics).

B.249 and B.250 follow from B.248. B.253 are determined by choice of σ_2' and operational semantics and choices made on maps related to the assertions.

B.238-B.241 follow from B.228-B.231 trivially by choices on maps of logical state and semantics of composition operators. B.236 follow from B.227, B.245-B.248 and choice of σ_2'. All compositions shown let us to derive conclusion for $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$. □

Lemma 21 (Alloc)

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

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

Proof: We assume

$$(\sigma, O, U, T, F) \ \in [\![\Gamma \ , x : \mathsf{undef} \ ]\!]_{M,tid} * \{m\}\rfloor) \tag{B.257}$$

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{B.258}$$

We split the composition in B.257 as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![\Gamma \ , x : \mathsf{undef} \ ]\!]_{M,tid} \tag{B.259}$$

$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{B.260}$$

$$\sigma_1 \bullet \sigma_2 = \sigma \tag{B.261}$$

$$O_1 \bullet_O O_2 = O \tag{B.262}$$

$$U_1 \cup U_2 = U \tag{B.263}$$

$$T_1 \cup T_2 = T \tag{B.264}$$

$$F_1 \uplus F_2 = F \tag{B.265}$$

$$\mathsf{WellFormed}(\sigma, O_1, U_1, T_1, F_1) \tag{B.266}$$

$$\mathsf{WellFormed}(\sigma, O_2, U_2, T_2, F_2) \tag{B.267}$$

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

$$(\sigma', O_1', U_1', T_1', F_1') \in [\![\Gamma\ , x : \mathsf{rcuFresh}\ \mathcal{N}_\emptyset]\!] \tag{B.268}$$

$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{B.269}$$

$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{B.270}$$

$$O_1' \bullet_O O_2' = O' \tag{B.271}$$

$$U_1' \cup U_2' = U' \tag{B.272}$$

$$T_1' \cup T_2' = T' \tag{B.273}$$

$$F_1' \uplus F_2' = F' \tag{B.274}$$

$$\mathsf{WellFormed}(\sigma', O_1', U_1', T_1') \tag{B.275}$$

$$\mathsf{WellFormed}(\sigma', O_2', U_2', T_2') \tag{B.276}$$

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

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

There exists no change in the observation of heap locations

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

B.279 follows from B.257

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

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(B.268) 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 B.275, B.276 and $(\sigma_1', O_1', U_1', T_1', F_1')$ and $(\sigma_2', O_2', U_2', T_2', F_2')$. To prove B.275, we need to show that each of the memory axioms in Section B.2 holds for the state $(\sigma', O_1', U_1', T_1', F_1')$.

Case 60 - ***UNQR*** *Determined by B.268 and operational semantics(ℓ is fresh-unique).*

Case 61 - ***RWOW***, ***OW*** *By B.268*

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 B.268.*

Case 65 - ***WF*** *By B.279, B.268 and B.278.*

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

Case 67 - ***FNR*** *By B.268 and operational semantics(ℓ is fresh-unique).*

Case 68 - ***FPI*** *By B.268 and $\mathcal{N}_{f,\emptyset}$.*

Case 69 - ***HD***

Case 70 - ***WNR*** *By B.279.*

To prove B.269, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{B.280}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{B.281}$$

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{B.282}$$

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{B.283}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{B.284}$$

$$\forall x, t \in T.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{B.285}$$

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h) \tag{B.286}$$

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h') \tag{B.287}$$

To prove all relations (B.280-B.287) we assume B.279 which is to assume T_2 as subset of reader threads and B.267. 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 B.267 and choices on maps we show B.276. B.282 and B.283 follow trivially. B.285 follows from choice of σ_2' and B.277(determined by operational semantics). B.271-B.274 follow from B.262-B.265, semantics of compositions operators and choices made for maps of the logical state.

B.280 and B.281 follow from B.279 and choice on F_2'. B.284 are determined by operational semantics, operational semantics and choices made on maps related to the assertion.

$\sigma_1'.h \cap \sigma_2'.h = \emptyset$ is determined by operational semantics(ℓ is unique and fresh). So, B.270 follows from B.261 and choice of σ_2'. All compositions shown let us to derive conclusion for $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$. □

Lemma 22 (Free)

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

$$\lfloor [\![x : \mathsf{undef}]\!] * \mathcal{R}(\{m\}) \rfloor$$

Proof: We assume

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

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{B.289}$$

We split the composition in B.288 as

$$(\sigma, O_1, U_1, T_1, F_1) \in [\![x : \mathsf{freeable}]\!]_{M,tid} \tag{B.290}$$

$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{B.291}$$

$$\sigma_1 \bullet \sigma_2 = \sigma \tag{B.292}$$

$$O_1 \bullet_O O_2 = O \tag{B.293}$$

$$U_1 \cup U_2 = U \tag{B.294}$$

$$T_1 \cup T_2 = T \tag{B.295}$$

$$F_1 \uplus F_2 = F \tag{B.296}$$

$$\mathsf{WellFormed}(\sigma, O_1, U_1, T_1, F_1) \tag{B.297}$$

$$\mathsf{WellFormed}(\sigma, O_2, U_2, T_2, F_2) \tag{B.298}$$

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

$$(\sigma', O_1', U_1', T_1', F_1') \in [\![x : \mathsf{undef}]\!] \tag{B.299}$$

$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{B.300}$$

$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{B.301}$$

$$O_1' \bullet_O O_2' = O' \tag{B.302}$$

$$U_1' \cup U_2' = U' \tag{B.303}$$

$$T_1' \cup T_2' = T' \tag{B.304}$$

$$F_1' \uplus F_2' = F' \tag{B.305}$$

$$\mathsf{WellFormed}(\sigma', O_1', U_1', T_1', F_1') \tag{B.306}$$

$$\mathsf{WellFormed}(\sigma', O_2', U_2', T_2', F_2') \tag{B.307}$$

From operational semantics we know that

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

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

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

$$U_1' = U_1 \cup \{(x, tid)\} \tag{B.311}$$

B.312 follows from B.288

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

Let T_1' to be T_1. All B.308-B.311 show(B.299) 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 B.306, B.307 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 B.306, we need to show that each of the memory axioms in Section B.2 holds for the state $(\sigma', O_1', U_1', T_1', F_1')$ and it it trivial by B.308-B.311 and B.299.

To prove B.300, 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

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{B.313}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{B.314}$$

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{B.315}$$

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{B.316}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.R = \sigma_2'.R \wedge \sigma_2.rt = \sigma_2'.rt \tag{B.317}$$

$$\forall x, t \in T.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{B.318}$$

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h) \tag{B.319}$$

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h') \tag{B.320}$$

To prove all relations (B.313-B.320) we assume B.312 which is to assume T_2 as subset of reader threads and B.267. 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 B.267 and choices on maps we show B.307. B.315 and B.316 follow trivially. B.318 follows from choice of σ_2' and B.308(determined by operational semantics). B.302-B.305 follow from B.294-B.296, semantics of composition operators and choices on related maps.

B.313 and B.314 follow from B.312 and choice on F_2'. B.317 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 B.302 straightforward. Since other machine components do not change(determined by operational semantics), B.301 follows from B.292, B.308 and choice of σ_2'. All compositions shown let us to derive conclusion for $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$. □

Lemma 23 (RReadStack)

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

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

Proof: We assume

$$(\sigma, O, U, T, F)\ \in [\![\Gamma\ , \Gamma\ , z : \mathsf{rcultr}\ , x : \mathsf{rcultr}]\!]_{R,tid} * \{m\} \tag{B.321}$$

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{B.322}$$

We split the composition in B.321 as

$$(\sigma_1, O_1, U_1, T_1, F_1) \in [\![\Gamma\ , \Gamma\ , z : \mathsf{rcultr}\ , x : \mathsf{rcultr}]\!]_{R,tid} \tag{B.323}$$

$$(\sigma, O_2, U_2, T_2, F_2) = m \tag{B.324}$$

$$O_1 \bullet_O O_2 = O \tag{B.325}$$

$$\sigma_1 \bullet \sigma_2 = \sigma \tag{B.326}$$

$$U_1 \cup U_2 = U \tag{B.327}$$

$$T_1 \cup T_2 = T \tag{B.328}$$

$$F_1 \uplus F_2 = F \tag{B.329}$$

$$\mathsf{WellFormed}(\sigma, O_1, U_1, T_1, F_1) \tag{B.330}$$

$$\mathsf{WellFormed}(\sigma, O_2, U_2, T_2, F_2) \tag{B.331}$$

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

$$(\sigma', O_1', U_1', T_1', F_1') \in [\![\Gamma\ , x : \mathsf{rcultr}\ , z : \mathsf{rcultr}]\!]_{R,tid} \tag{B.332}$$

$$(\sigma', O_2', U_2', T_2', F_2') \in \mathcal{R}(\{m\}) \tag{B.333}$$

$$O_1' \bullet_O O_2' = O' \tag{B.334}$$

$$\sigma_1' \bullet \sigma_2' = \sigma' \tag{B.335}$$

$$U_1' \cup U_2' = U' \tag{B.336}$$

$$T_1' \cup T_2' = T' \tag{B.337}$$

$$F_1' \uplus F_2' = F' \tag{B.338}$$

$$\mathsf{WellFormed}(\sigma', O_1', U_1', T_1', F_1') \tag{B.339}$$

$$\mathsf{WellFormed}(\sigma', O_2', U_2', T_2', F_2') \tag{B.340}$$

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

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

There exists no change in the observation of heap locations

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

B.343 follows from B.321

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

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 B.323 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 B.339, B.340B.24, B.25 and show the composition of $(\sigma_1', O_1', U_1', T_1', F_1')$ and $(\sigma_2', O_2', U_2', T_2', F_2')$. To prove B.339, we need to show that each of the memory axioms in Section B.2 holds for the state $(\sigma', O_1', U_1', T_1', F_1')$ which is trivial by assuming B.330 and knowing B.343, B.342 and components of the state determined by operational semantics.

To prove B.340, we need to show that WellFormedness is preserved under interference relation

$$(\sigma, O_2, U_2, T_2, F_2)\mathcal{R}(\sigma', O_2', U_2', T_2', F_2')$$

which by definition means that we must show

$$\sigma_2.l \in T_2 \rightarrow (\sigma_2.h = \sigma_2'.h \wedge \sigma_2.l = \sigma_2'.l) \tag{B.344}$$

$$l \in T_2 \rightarrow F_2 = F_2' \tag{B.345}$$

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2.h) \tag{B.346}$$

$$\forall tid, o.\, \mathsf{iterator}\ tid \in O_2(o) \rightarrow o \in dom(\sigma_2'.h) \tag{B.347}$$

$$O_2 = O_2' \wedge U_2 = U_2' \wedge T_2 = T_2' \wedge \sigma_2.B = \sigma_2'.B \wedge \sigma_2.rt = \sigma_2'.rt \tag{B.348}$$

$$\forall x, t \in T_2.\, \sigma_2.s(x,t) = \sigma_2'.s(x,t) \tag{B.349}$$

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h) \tag{B.350}$$

$$\forall tid, o.\, \mathsf{root}\ tid \in O(o) \rightarrow o \in dom(h') \tag{B.351}$$

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 B.344-B.351 trivial and B.334-B.337 follow from assumptions B.325-B.329, choices made for related maps and semantics of composition operations. All compositions shown let us derive conclusion for $(\sigma_1', O_1', U_1', T_1', F_1') \bullet (\sigma_2', O_2', U_2', T_2', F_2')$. □

Lemma 24 (RReadHeap)

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

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

Proof: We assume

$$(\sigma, O, U, T, F)\ \in [\![\Gamma\ , z : \mathsf{rcultr}\ , x : \mathsf{rcultr}]\!]_{R,tid} * \{m\}\rfloor) \tag{B.352}$$

$$\mathsf{WellFormed}(\sigma, O, U, T, F) \tag{B.353}$$

We split the composition in B.352 as

$$(\sigma_1, O_1, U_1, T_1, F_1) \in [\![\Gamma\ , z : \mathsf{rcultr}\ , x : \mathsf{rcultr}]\!]_{R,tid} \tag{B.354}$$

$$(\sigma_2, O_2, U_2, T_2, F_2) = m \tag{B.355}$$

$$\sigma_1 \bullet \sigma_2 = \sigma \tag{B.356}$$

$$O_1 \bullet_O O_2 = O \tag{B.357}$$

$$U_1 \cup U_2 = U \tag{B.358}$$

$$T_1 \cup T_2 = T \tag{B.359}$$

$$F_1 \uplus F_2 = F \tag{B.360}$$

$$\mathsf{WellFormed}(\sigma_1, O_1, U_1, T_1, F_1) \tag{B.361}$$

$$\mathsf{WellFormed}(\sigma_2, O_2, U_2, T_2, F_2) \tag{B.362}$$

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

$$(\sigma_1', O_1', U_1', T_1', F_1') \in \lfloor [\![\Gamma\ , x : \mathsf{rcultr}\ , z : \mathsf{rcultr}]\!] \tag{B.363}$$

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

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

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

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

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

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

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

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

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

There exists no change in the observation of heap locations

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

B.373 follows from B.352

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

Proof is similar to Lemma 23. □

B.4 Soundness Proof of Structural Program Actions

In this section, we introduce soundness Theorem B.4.1 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 B.4.1 (Type System Soundness)

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

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

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

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

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

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

$$\{[\![\Gamma']\!]_{M,tid}\} C \{[\![\Gamma'']\!]_{M,tid}\} \tag{B.377}$$

We need to show

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

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

From B.375 and Lemma 25 we know

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

From B.376 and 25 we know

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

By using B.379, B.380 and B.377 as antecedentes of Views Logic's consequence rule, we conclude B.378.

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

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

We know

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

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

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

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

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

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

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

$$\Gamma(x) = \mathit{bool} \tag{B.387}$$

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

Our goal is to prove

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

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

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

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

The poof of B.390 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 B.388

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

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

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

to the proof rule for nondeterministic looping.

The proof of B.391 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* $(e)\,then\{C_1\}else\{C_2\}$.

$$\Gamma, x : \mathit{rcultr}\,\rho\,\mathcal{N}([f_1 \rightharpoonup z]) \vdash_M C_1 \dashv \Gamma' \tag{B.392}$$

$$\Gamma, x : \mathit{rcultr}\,\rho\,\mathcal{N}([f_2 \rightharpoonup z]) \vdash_M C_2 \dashv \Gamma' \tag{B.393}$$

$$\{[\![\Gamma,\ x : \mathit{rcultr}\,\rho\,\mathcal{N}([f_1 \rightharpoonup z])]\!]_{M,tid}\} C_1 \{[\![\Gamma']\!]_{M,tid}\} \tag{B.394}$$

$$\{[\![\Gamma,\ x : \mathit{rcultr}\,\rho\,\mathcal{N}([f_2 \rightharpoonup z])]\!]_{M,tid}\} C_2 \{[\![\Gamma']\!]_{M,tid}\} \tag{B.395}$$

Our goal is to prove

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

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 proveB.396 from the consequence rule of Views Logic

based on the proofs of the following B.397 and B.398

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

and

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

B.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 B.397 from the branch rule of Views Logic based on the proofs of the following B.399 and B.400

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

and

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

We show B.399 from Views Logic's proof rule for the assume construct by using

$$\{[\![\Gamma, x : \mathit{rcultr}\ \rho\ \mathcal{N}([f_1|f_2 \rightharpoonup z])]\!]_{M,tid} \cap$$

$$[\![x : \mathit{rcultr}\ \rho\ \mathcal{N}([f_1 \rightharpoonup y])]\!]_{M,tid}\}$$

$$assume\,(y = z)$$

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

as the antecedent. We can use this antecedent together with

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

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

We show B.400 from Views Logic's proof rule for the assume construct by using

$$\{[\![\Gamma, x : \mathit{rcultr}\ \rho\ \mathcal{N}([f_1|f_2 \rightharpoonup z]) \cap$$

$$x : \mathit{rcultr}\ \rho\ \mathcal{N}([f_1 \rightharpoonup y])]\!]_{M,tid}\}$$

$$assume(x \neq y)$$

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

as the antecedent. We can use this antecedent together with

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

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*

$$\Gamma_1 \vdash C_1 \dashv \Gamma_1' \tag{B.401}$$

$$\Gamma_2 \vdash C_2 \dashv \Gamma_2' \tag{B.402}$$

$$\{[\![\Gamma_1]\!]\}C_1\{[\![\Gamma_1']\!]\} \tag{B.403}$$

$$\{[\![\Gamma_2]\!]\}C_2\{[\![\Gamma_2']\!]\} \tag{B.404}$$

We need to show

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

By using B.403 and B.404 as antecedents to Views Logic's parallel rule, we can draw conclusion for B.406

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

Showing B.405 requires showing

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

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

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

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

$$\Gamma, y : \mathsf{rcultr}\ \_ \vdash_M C \dashv \Gamma' \tag{B.409}$$

$$\mathsf{FType}(f) = \mathsf{RCU} \tag{B.410}$$

$$\mathsf{NoFresh}(\Gamma') \tag{B.411}$$

$$\mathsf{NoFreeable}(\Gamma') \tag{B.412}$$

$$\mathsf{NoUnlinked}(\Gamma') \tag{B.413}$$

$$\{[\![\Gamma\ , y : \mathsf{rcultr}\ \_]\!]_{M,tid}\} C \{[\![\Gamma']\!]_{M,tid}\} \tag{B.414}$$

Our goal is to prove

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

Any case of C does not change the state(no heap update) by assumptions B.411-B.413 therefore B.415 follows from assumptions B.409-B.414 trivially.

□

Lemma 25 (Context-SubTyping-M)

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

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

Lemma 26 (Context-SubTyping-R)

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

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

Lemma 27 (Singleton-SubTyping-M)

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

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

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

together with the approximation on the field map

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

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

Lemma 28 (Singleton-SubTyping-R)

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

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

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

□

APPENDIX C

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

$parent = current;$

$\left\{parent : rcuItr\ (Left|Right)^k\ \{\}\right\}$

$\left\{current : rcuItr\ (Left|right)^k\ \{\}\right\}$

$current = current.Right;$

$\left\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto current\}\right\}$

$\left\{current : rcuItr\ (Left|Right)^k.Right\ \{\}\right\}$

}

}

$\{parent : rcuItr\ (Left|Right)^k\ \{(Left|Right) \mapsto current\}\}$

$\{current : rcuItr\ (Left|Right)^k.(Left|Right)\ \{\}\}$

// At this point, we've found the node to remove

$BinaryTreeNode\ lmParent = current.Right;$

$BinaryTreeNode\ currentL = current.Left;$

$\{current : rcuItr(Left|Right)^k.(Left|Right)\{Left \mapsto currentL, Right \mapsto lmParent\}\}$

$\{currentL : rcuItr\ (Left|Right)^k.(Left|Right).Left\ \{\}\}$

$\{lmParent : rcuItr\ Left|Right)^k.(Left|Right).Right\ \{\}\}$

// We now need to "rethread" the tree

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

$if\ (current.Right == null)$

{

$\{parent : rcuItr\ (Left|Right)^k\ \{(Left|Right) \mapsto current\}\}$

$\{current : rcuItr\ (Left|Right)^k.(Left|Right)\ \{Left \mapsto currentL, Right \mapsto null\}\}$

$\{currentL : rcuItr\ (Left|Right)^k.(Left|Right).Left\ \{\}\}$

$if\ (parent.Left == current)$

// parent.Value 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

$else\ if\ (current.Left == null)$

{

$\{parent : rcuItr\ (Left|Right)^k\ \{(Left|Right) \mapsto current\}\}$

$\{current : rcuItr\ (Left|Right)^k.(Left|Right)\ \{Left \mapsto null, Right \mapsto lmParent\}\}$

$\{currentL : rcuItr\ (Left|Right)^k.(Left|Right).Left\ \{\}\}$

$\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right\ \{\}\}$

$if\ (parent.Left == current)$

$\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto current\}\}$

$\{current : rcuItr\ (Left|Right)^k.Left\ \{Left \mapsto null, Right \mapsto lmParent\}\}$

$\{lmParent : rcuItr\ (Left|Right)^k.Left.Right\ \{\}\}$

// parent.Value is greater than current.Value

// so make current's right child a left child of parent

$parent.Left = lmParent;$

$\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto lmParent\}\}$

$\{current : unlinked\}$

$\{lmParent : rcuItr\ (Left|Right)^k.Left\ \{\}\}$

$else$

$\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto current\}\}$

$\{current : rcuItr\ (Left|Right)^k.Right\ \{Left \mapsto null, Right \mapsto lmParent\}\}$

$\{lmParent : rcuItr\ (Left|Right)^k.Right.Right\ \{\}\}$

// parent.Value is less than current.Value

// so make current's right child a right child of parent

$parent.Right = lmParent;$

$\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto lmParent\}\}$

$\{lmParent : rcuItr\ (Left|Right)^k.Right\ \{\}\}$

$\{current : unlinked\}$

$SyncStart;\ SyncStop;$

$\{current : freeable\}$

$Free(current);$

$\{current : undef\}$

}

// CASE 3: If current's right child has a left child, replace current with current's

// right child's left-most descendent

$else$

{

$\{parent : rcuItr\ (Left|Right)^k\ \{(Left|Right) \mapsto current\}\}$

$\{current : rcuItr\ (Left|Right)^k.(Left|Right)\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$

$\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right\ \{\}\}$

$\{currentL : rcuItr\ (Left|Right)^k.(Left|Right).Left\ \{\}\}$

// 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

$parent.Left = currentF;$

$\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto currentF\}\}$

$\{current : unlinked\}$

$\{currentF : rcuItr\ (Left|Right)^k.Left\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$

$SyncStart;\ SyncStop;$

$\{current : freeable\}$

$Free(current);$

$\{current : undef\}$

}

$else\{$

$\left\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto current\}\right\}$

$\left\{current : rcuItr\ (Left|Right)^k.Right\ \{Right \mapsto lmParent, Left \mapsto currentL\}\right\}$

$\{currentF : rcuFresh\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$

//current's right child a right child of parent

$parent.Right = currentF;$

$\left\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto currentF\}\right\}$

$\{current : unlinked\}$

$\left\{currentF : rcuItr\ (Left|Right)^k.Right\ \{Right \mapsto lmParent, Left \mapsto currentL\}\right\}$

$SyncStart;\ SyncStop;$

$\{current : freeable\}$

$Free(current);$

$\{current : undef\}$

}

} $else\{$

$\left\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right\ \{Left \mapsto leftmost\}\right\}$

$\left\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left\ \{\}\right\}$

$while\ (leftmost.Left! = null)$

{

$\left\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l\ \{Left \mapsto leftmost\}\right\}$

$\left\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{\}\right\}$

$lmParent = leftmost;$

$\left\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{\}\right\}$

$\left\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{\}\right\}$

$leftmost = lmParent.Left;$

$\left\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{Left \mapsto leftmost\}\right\}$

$\left\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left.Left\ \{\}\right\}$

}

$currentF.data = leftmost.data;$

$if\ (parent.Left == current)\{$

$\left\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto current\}\right\}$

$\left\{current : rcuItr\ (Left|Right)^k.Left\ \{Right \mapsto lmParent, Left \mapsto currentL\}\right\}$

$\{currentF : rcuFresh\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$

//current's right child a left child of parent

$parent.Left = currentF;$

$\left\{parent : rcuItr\ (Left|Right)^k\ \{Left \mapsto currentF\}\right\}$

$\{current : unlinked\}$

$\left\{currentF : rcuItr\ (Left|Right)^k.Left\ \{Right \mapsto lmParent, Left \mapsto currentL\}\right\}$

$SyncStart;\ SyncStop;$

$\{current : freeable\}$

$Free(current);$

$\{current : undef\}$

}

$else\{$

$\left\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto current\}\right\}$

$\left\{current : rcuItr\ (Left|Right)^k.Right\ \{Right \mapsto lmParent, Left \mapsto currentL\}\right\}$

$\{currentF : rcuFresh\ \{Right \mapsto lmParent, Left \mapsto currentL\}\}$

//current's right child a right child of parent

$parent.Right = currentF;$

$\left\{parent : rcuItr\ (Left|Right)^k\ \{Right \mapsto currentF\}\right\}$

$\{current : unlinked\}$

$\left\{currentF : rcuItr\ (Left|Right)^k.Right\ \{Right \mapsto lmParent, Left \mapsto currentL\}\right\}$

$SyncStart;\ SyncStop;$

$\{current : freeable\}$

$Free(current);$

$\{current : undef\}$

}

$\left\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l\ \{Left \mapsto leftmost\}\right\}$

$\left\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{Left \mapsto null\}\right\}$

$BinaryTreeNode\ leftmostR = leftmost.Right;$

$$\left\{leftmost : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \left\{\begin{array}{l} Left \mapsto null, \\ Right \mapsto leftmostR \end{array}\right\}\right\}$$

$\left\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l\ \{Left \mapsto leftmost\}\right\}$

$\left\{leftmostR : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left.Right\ \{\}\right\}$

// the parent's left subtree becomes the leftmost's right subtree

$lmParent.Left = leftmostR;$

$\{leftmost : unlinked\}$

$\left\{lmParent : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l\ \{Left \mapsto leftmostR\}\right\}$

$\left\{leftmostR : rcuItr\ (Left|Right)^k.(Left|Right).Right.Left(Left)^l.Left\ \{\}\right\}$

$SyncStart;\ SyncStop;$

$\{leftmost : freeable\}$

$Free(leftmost);$

$\{leftmost : undef\}$

}

}

$WriteEnd;$

}

APPENDIX D

RCU BAG WITH LINKED-LIST

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

$void\ remove\ (int\ toDel\ )\ \{$

$WriteBegin;$

$BagNode\ current,\ parent\ =\ head;$

$current\ =\ parent.Next;$

$\{current : rcuItr\ Next\ \{\}\}$

$\{parent : rcuItr\ \epsilon\ \{Next \mapsto current\}\}$

$while\ (current.Next! = null \&\& current.data \neq toDel)\ \{$

$\left\{parent : rcuItr\ (Next)^k\ \{Next \mapsto current\}\right\}$

$\left\{current : rcuItr\ Next.(Next)^k.Next\ \{\}\right\}$

$parent = current;$

$\left\{current : rcuItr\ Next.(Next)^k.Next\ \{\}\right\}$

$\left\{parent : rcuItr\ Next.(Next)^k.Next\ \{\}\right\}$

$current = parent.Next;$

$\left\{parent : rcuItr\ Next.(Next)^k.Next\ \{Next \mapsto current\}\right\}$

$\left\{current : rcuItr\ Next.(Next)^k.Next.Next\ \{\}\right\}$

}

//We don't need to be precise on whether next of current is null or not

$\left\{parent : rcuItr\ Next.(Next)^k.Next\ \{Next \mapsto current\}\right\}$

$\left\{current : rcuItr\ Next.(Next)^k.Next.Next.Next\ \{Next \mapsto null\}\right\}$

$BagNode\ currentL = current.Next;$

$\left\{parent : rcuItr\ Next.(Next)^k.Next\ \{Next \mapsto itr\}\right\}$

$\left\{currentL : rcuItr\ Next.(Next)^k.Next.Next.Next\ \{\}\right\}$

$\left\{current : rcuItr\ Next.(Next)^k.Next.Next\ \{Next \mapsto currentL\}\right\}$

$current.Next = currentL;$

$\left\{parent : rcuItr\ Next.(Next)^k.Next\ \{Next \mapsto itrN\}\right\}$

$\left\{currentL : rcuItr\ Next.(Next)^k.Next.Next\ \{\}\right\}$

$\{current : unlnked\}$

$SyncStart;$

$SyncStop;$

$\{current : freeable\}$

$Free(current);$

$\{current : undef\}$

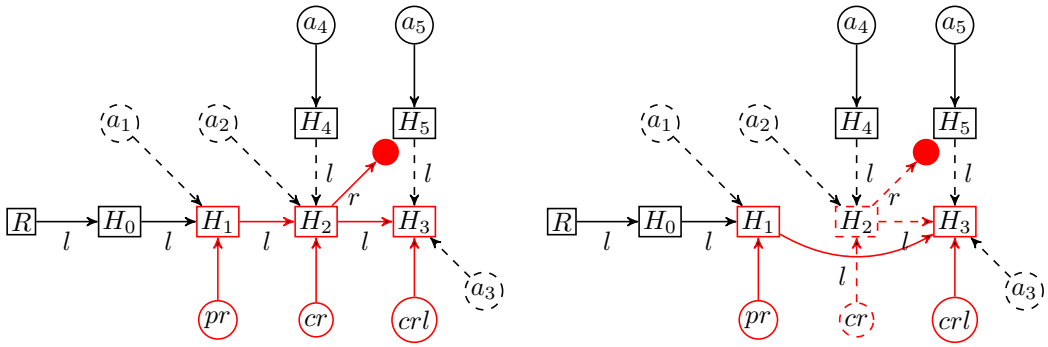
$WriteEnd;$

}

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

APPENDIX E

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

Figure E.1 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 E.1a and E.1b. 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 E.1a 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 E.1a and E.1b. 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 E.1a to Figure E.1b. 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 E.1a. When we look at Figure E.1b, 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 rculltr type.

One another safety achieved with framing shows up in a setting where `current` and a_2 are aliases. In the Figure E.1a, both `current` and a_2 are in the rculltr 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 rculltr 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 E.1a and E.1b: 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 E.1a and E.1b.

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 E.1b.

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

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

APPENDIX F

TYPES RULES FOR RCU READ SECTION

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

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

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

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

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

$$\frac{\Gamma_1 \vdash C_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash C_2 \dashv \Gamma_3}{\Gamma_1 \vdash C_1 \;;\; C_2 \dashv \Gamma_3} \text{(T-Seq)}$$

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

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

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

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

Figure F.1 Type Rules for Read critical section for RCU Programming