Verifying Concurrent Systems Code

Travis Hance

CMU-CS-CS-24-146

August 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Bryan Parno (Chair)
Dave Andersen
Frank Pfenning
Derek Dreyer (Max Planck Institute for Software Systems)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

The work presented in this thesis was supported in part by the Alfred P. Sloan Foundation, the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award No. CNS-1700521, National Science Foundation grant CCF-2318953, a Google Faculty Fellowship, a grant from the Intel Corporation, a gift from VMware, an internship at VMware, an Amazon Research Award (Fall 2022 CFP), and the Future Enterprise Security initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab). Any opinions, findings, conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of these sponsors.

Keywords: Rust, systems, formal verification, separation logic, Iris, type systems, ownership types, semantic types

For my parents, Darrell and Sandra Hance

Abstract

Concurrent software is notoriously difficult to write correctly, so to increase confidence in it, it is often desirable to apply formal verification techniques. One technique that is especially promising for verifying concurrent software is *concurrent separation logic (CSL)*, which uses reasoning principles based on resource ownership. However, even with CSL, verifying complex systems at scale (e.g., those with 1000s of lines of code) remains challenging. The reasons it remains challenging include,

(1) The manual proof effort required by many existing CSL frameworks.
(2) The inherent complexity of the target systems. Sophisticated systems may have custom, low-level synchronization logic, which may be deeply intertwined with domain logic, in the interest of performance.

We posit that a promising way to overcome (1) is, rather than using CSL directly, to use an ownership type system such as Rust's, taking advantage of its sophisticated but efficient type-checking algorithms. To demonstrate this, we develop a full methodology, from theory to implementation, based around this core idea, showing that we can recover the rich reasoning principles of CSL in this setting. In particular, we show that this methodology is rich enough to support the verification of inherently complex systems as in (2).

Acknowledgments

This thesis would not be possible without the support of my mentors, colleagues, friends, and family.

Firstly, I have to thank my advisor, Bryan Parno, for his tremendous support and guidance, Bryan always believed in me, pushed me, and supported the directions I wanted to pursue, even when they appeared to be moonshots. Bryan taught me how to write (put up with my drafts) and how to give talks (put up with my practice talks), and just generally how to select and pursue a research direction. And he did all of that with a lab of nine students.

I also need to thank my other mentors and colleagues through the years. I'd like to thank the members of my thesis committee, Frank Pfenning, Dave Andersen, and Derek Dreyer for all their helpful feedback. I thank Jon Howell and Rob Johnson, my intern mentors at VMware, for a memorable summer that kicked off many of my productive research threads. For Rob, I thank you for rekindling my old interest in high-performance data structures, and for Jon, I thank you for showing me how to think about the developer experience in verification.

I'd also like to thank the Verus contributors for making it a fun and productive research project, including Andrea Lattuada, Chris Hawblitzel, Chanhee Cho, Jay Bosamiya, Yi Zhou, Jay Lorch, Reto Achermann, Tej Chajed, and others. I'd also like to thank my various other collaborators over the years, including Oded Padon, Ruben Martins, Marijn Huele, Alex Conway, Ryan Stutsman, and Gerd Zellweger. So much work was made possible because of all of you. I'd also like to thank Jean Yang for encouraging me to apply to graduate school in the first place.

Finally, I'd like to thank my friends and family. Most of all this includes my parents, Sandra and Darrell, without whom I certainly would not be here. I'd like to thank my brother, Jared, as well, and all of my friends who helped kept me sane during the pandemic. Last but not least, I'd like to thank Michael Cohen, who remains dear in my memory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The challenge of verified systems software

For decades, formal verification has promised to deliver correct, bug-free software, yet today, most software remains unverified, in part because of the massive amount of effort involved in the endeavor. This is especially true of low-level systems software, which is challenging for a variety of reasons: it is often ruthlessly optimized, it frequently involves multi-threaded code with complex interleavings, and it tends to lack the abstractions that we often rely on in higher-level programming languages.

What makes software so hard to reason about? There are a number of reasons, but there is one that always makes itself known quickly. Consider the following code:

```
void example(int* a, int *b) {
    *a = 0x07151129;
    foo(b);
    int x = *a;
}
```

Will `x` necessarily have the value 0x07151129? Well, it depends. Maybe `foo(b)` doesn't modify anything, so the answer is yes. Maybe it modifies some data structure that `b` points to, which might point to the same integer that `a` points to, so the answer is no. Or maybe `a` points to a global value that `foo` also writes to.

Now consider this:

```
void example2(int* a) {
    *a = 0x07151129;

    int x = *a;
}
```

Again: Will `x` necessarily have the value 0x07151129? Surely, *now* the answer is "yes." Or is it? What if there's another *thread* running simultaneously with this one which also has access to the `a` pointer?

Fundamentally, the reason these programs are hard to reason about is that we have provided no way to answer the question: *"Who else is doing who knows what with my data?"*

Luckily, all is not lost, and researchers are increasingly turning to a certain reasoning principle to solve this particular problem. The reasoning principle is called ***ownership***. The basic idea is to have a way to ensure some kind of restrictions that helps us avoid these situations where "others" can access or modify our data in unexpected ways. Ownership is an idea that has its foothold in multiple areas along the software reasoning stack, both in formal program logics and in type systems adopted by mainstream programming languages.

Among program logics, one of the most successful has been *concurrent separation logic (CSL)* [61]. CSL allows proof developers to reason about ownership over memory permissions and other abstract resources via a rich resource logic. Modern CSLs such as Iris [35] have been deployed for numerous challenging problems and they can address a variety of deep program properties. However, CSL remains technically challenging to learn and use, and it can require an enormous amount of effort to deploy it at scale.

On the other hand, then, what can we say of more "mainstream" applications of ownership reasoning? A recent success story in programming language design is found in the *Rust* programming language [54], which uses an ownership-based type system to ensure memory safety without a garbage collector, and as a result, is widely regarded as an excellent choice for systems programming. The type system even has benefits for formal verification, and a number of Rust verification tools have made their appearance, taking advantage of this fact [2, 18, 30].

However, Rust's type system also has a caveat. Its type system is sometimes insufficiently flexible for certain situations, and the developer has to use its infamous *unsafe code*—opting in to more flexibility, but opting out of the safety guarantees, putting more responsibility and more risk on the programmer. Even elementary data structures like *doubly-linked lists* fall outside of Rust's safe fragment, let alone the bleeding-edge data structures that appear in modern systems code. The *conventional wisdom*, at least, is that the benefits of Rust's ownership type system do not extend to this kind of code.

So where we will find our solution to realistic systems verification? How can we handle both the *complexity* and *scale* of realistic systems—that is, how do we reason through the subtle correctness arguments of intricate, carefully-developed optimizations, even when those arguments cut across codebases measured in the 1000s of lines or more? Will the solution be in the rich-but-difficult program logics, or in the efficient-but-inflexible type systems?

1.2 My thesis

We can have the best of both worlds, simultaneously utilizing the scalable ownership-based type system while recovering the rich reasoning principles of modern CSL to tackle challenging algorithms.

In this document, I will develop a methodology to do exactly this. Specifically, I will show how to adapt these rich reasoning principles into an automated verification tool that takes advantage of an ownership-based type system for efficiency and productivity. Notably, this is not *quite* so simple as copy-and-pasting some laws from CSL as axioms into our automated verification tool, for it turns out that these laws have nontrivial interactions with the advanced features of the

ownership type system. Fortunately, working out how to handle these interactions properly leads to rewarding new proof styles.

I have implemented the methodology across two tools in total: first, the *IronSync* framework in *Linear Dafny*, and the *Verus* tool for Rust verification. The second tool is a continuation of the first, so it will be the primary focus of the thesis.

However, I will evaluate both tools. To do this, we will use four case studies, one of which is implemented in both. The studies include three "major" case studies, each around 1000 lines of code of more, and each representing a complex, specialized piece of concurrent systems software. Finally, we include one "minor" case study, representing a much smaller but ubiquitous library utility.

1.3 The events leading to this development

I want to talk a bit about how all this was developed, both to make the motivations a bit more concrete, and to highlight the contributions from other key players.

I was always interested in verifying practical systems software, and as a result I ended up in an internship at VMware Research Group, working with Jon Howell, Rob Johnson, and a fellow intern, Andrea Lattuada, to verify a disk-backed key-value storage system in Dafny, a project that would come to be known as VeriBetrKV [25]. The main focus of VeriBetrKV was on verifying crash-safety properties; this turned out to be pretty interesting, though unfortunately it will not be the focus of this document. The reason is that, throughout our work, we began to struggle with a different issue. Specifically, we found that it was very difficult to work with mutable data structures because doing so required us to write very complicated conditions about pointer aliasing. This is the same challenge I described at the beginning of this chapter.

Jon and Andrea had an intuition that an ownership type system was the way forward, and they were specifically interested in Rust's type system. At the time (2019), the Rust verification world was still fledgling, but we kept the idea in the back of our heads. As the VeriBetrKV project reached feature-completion, we turned towards integrating an ownership type system into Dafny, something like a "Rust-lite" type system. Bryan Parno and Chris Hawblitzel had also discussed the idea before based on their exprience with IronFleet [29]; Chris became involved with VeriBetrKV and led the development of *Linear Dafny*, which ultimately had many of the benefits we'd theorized [51].

Meanwhile, I was somewhat dissatisfied with the single-threaded nature of our approach in VeriBetrKV, so I turned my attention to concurrent verification, starting with a multi-threaded page cache called SplinterCache (now one of the case studies in this thesis) [11]. After catching up to the state-of-the-art in concurrent verification, I concluded that concurrent separation logic was the way forward. Separation logic was also based on ownership, so it had good synergy with Linear Dafny's ownership types. Furthermore, from the *Iris* separation logic I learned of a formal mathematical concept called a *resource algebra* [31, 32] which seemed to be suitably general. I worked to integrate resource algebras into Linear Dafny in what would become the *IronSync* framework [28].

Though the direction was promising, I was initially stumped because there was no obvious way to do handle the necessary interactions with Linear Dafny's shared references. Still focused

on SplinterCache, I was motivated to develop a new algebraic structure, loosely based on the resource algebra, that would help me verify SplinterCache's reader-writer lock. These laws were the first draft of (what would eventually be called) my *storage protocol* concept [26]. At first, I worried that the storage protocol concept was *ad hoc* and too narrowly useful. However, I later became involved with the effort to verify Node Replication, another one of our case studies, and I found that the ideas were applicable there as well. This convinced me of the concept's generality and usefulness, though it would take some years to justify the storage protocol with satisfactory formal footing, a process that involved a couple of false starts.

Now, despite all that we were accomplishing with Linear Dafny, we were becoming increasingly convinced that we were reaching its limits. We were interested in the verification of high-performance software systems, but Dafny had never really been intended as a high-performance systems language. Limitations on Linear Dafny's type system made us eager for a more fleshed out ownership type system like Rust's. On top of that, the piles and piles of hacks that composed Linear Dafny and IronSync made it difficult to progress forward on the tooling. It was thus that Chris pushed for a "clean slate" approach: to write a verification-condition-generation pipeline from scratch, consolidating lessons from prior tools and eliminating baggage. This project became Verus [42, 44], the verification tool for Rust, and the others soon got involved.

As the IronSync project wound to its conclusion, I shifted my focus onto Verus development as well, becoming a core contributor and integrating IronSync's ideas into Verus. With Rust's more advanced ownership type system, I was able to flesh out the ideas even further, and I continued working to simplify proof effort. I developed a novel framework, VerusSync, to abstract away many of the frustrations we had in IronSync, and as a result, I was able to take on a project even more sophisticated than either of the two IronSync case studies: a concurrent memory allocator. As a result, I am convinced that Verus, and more generally, the common methodology between IronSync and Verus, is a practical basis for systems verification. And this brings us to this thesis.

1.4 Publications

The content of this thesis is primarily based on the following publications, on each of which I am the first or second author:

- *Storage Systems are Distributed Systems (So Verify Them That Way!)* [25]
- *Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems* [28]
- *Leaf: Modularity for Temporary Sharing in Separation Logic* [26]
- *Verus: Verifying Rust Programs Using Linear Ghost Types* [42]
- *Verus: A Practical Foundation for Systems Verification* [44]

However, this thesis expands and unifies the presentation into a more coherent narrative than has appeared previously. Furthermore, some technical content, such as that appearing in Chapter 5 and Chapter 6 has not yet appeared in publication.

1.5 Thesis structure

In Chapter 2, I will introduce our four case studies in order to give the reader an idea of the shape of the problem facing us. In Chapter 3, I will introduce our methodology through our Rust verification tool, Verus. This chapter will highlight the key components, keeping the discussion informal, and go through several examples. In Chapter 4 – Chapter 7, I will characterize the methodology in more formal detail. In Chapter 8, I will briefly cover the essential differences between Verus and Linear Dafny and discuss the evolution of the methodology. In Chapter 9, I will return to the case studies, cover how our methodology is able to tackle all of them, and evaluate them. Finally, I will compare to related work (Chapter 10) and conclude (Chapter 11).

Chapter 2

Motivation and Case Studies

2.1 Case Study Overview

In this chapter, we look at main case studies that are the subject of this thesis and examine key design decisions. The primary objective of this section is to identify the **key challenges** that make the system difficult to reason about, either from an informal perspective or a formal one (in most cases, both).

Note that our objective is *not* to justify or evaluate the design decisions themselves beyond what is necessary to explain them. We take it as a given that the systems are well-designed and realistic since this has already been demonstrated by their authors in their original publications. Rather, the aim of this chapter is to see what challenges arise in formally reasoning about realistic systems that were designed *without formal verification* in mind.

Finally, note that I was not involved in the original implementations of any of these systems; furthermore, the descriptions and analyses are based on my own observations.

2.2 Scope and the correctness properties of interest

Before diving in, allow me to delineate the scope of the correctness properties we are going to be considering.

There are a number of properties one might be concerned with proving about a given program or system:

1. *Safety*, that is, that the program does not exhibit some undesirable behavior. Safety can include:
    - Freedom from memory safety violations (spatial memory safety or temporal memory safety)
    - Freedom from data races
    - Freedom from panics and early exits
2. *Functional correctness*, that is, any outputs or observable events of the program correspond to some *specification.*

3. *Liveness*, properties like "the program will terminate" or "some desired thing will eventually happen."
4. *Leak-freedom*, the absence of memory leaks or of other resource leaks.
5. *Privacy-preservation*, that the observable properties are independent of some secret data. This is an example of a *hyperproperty*, a property over sets of possible executions rather than a property over a single execution.

Properties of interest The main focus of this thesis is on *safety* and *functional correctness* properties. The two are in fact closely related: We cannot even attempt to prove functional correctness without safety, and safety is sometimes (though not always) reliant on functional correctness. Specifically, for safety, we are concerned the most with the absence of *undefined behavior*, which includes both memory safety and data-race-freedom. For functional correctness, the exact specification depends on the application.

Properties that are out of scope We treat liveness and termination as completely out-of-scope. Termination for concurrent programs is in general quite difficult, and we do not attempt it here. We also do not look at leak-freedom or any hyperproperties.

We also do not address panic-freedom; in fact, some of the code we will consider intentionally panics in certain situations. In principle, panic-freedom isn't much different than other kinds of safety violations, though it might be more challenging to eliminate all panics (as panics are often the "final escape hatch" from situations that are hard to reason about).

2.3 Case Study I: SplinterCache

SplinterCache is a multi-threaded, in-memory page cache that manages disk access for the key-value store SplinterDB [11]. SplinterDB is a high-performance key-value store, which can achieve a throughput of 3 M ops/sec. across 16 threads; SplinterCache is a crucial component of SplinterDB's operation. SplinterCache can handle cache sizes at a range between 4 GiB and upwards of 100 GiB, and it targets low-latency IO devices. It is ~ 2000 lines of C.

Though SplinterCache was designed specifically for SplinterDB, it has a fairly general-purpose interface, allowing the client to request a lock (read-only or writable) on a given 4 KiB disk page, which SplinterCache will load in from disk if necessary. Internally, SplinterCache is responsible for deciding which pages to evict and when. The cache uses asynchronous writeback, so it also has to ensure that data is written before eviction occurs. It uses batched IO when possible, and it supports an interface for the client to request prefetching. It uses a "clock" eviction policy [13].

2.3.1 Challenges

Elements like the eviction policy and IO batching are critical for performance and introduce a lot of complexity, but in fact, these mostly takes the form of high-level decision logic that is not

so relevant for the safety and correctness proofs. If we were to consider liveness, these elements might be more relevant, but we will not discuss them much here.

The most important elements for safety and correctness are the *locking mechanism*, the *cache page data*, and the *cache metadata*.

The cache page data refers to the collection of cache entries themselves, the memory that mirrors the disk contents, while **the cache metadata** manages the relationship between in-memory cache entries and disk pages. Specifically, it tracks which cache entry is assigned to which disk page, and whether each cache entry has recent changes that must be written back to disk. At the data level, this information is represented via:

- An entry, per disk page, that maps it to a cache entry (if any)
- An entry, per cache entry, that maps it to a disk page (if any)
- A status field, per cache entry, with a number of bit flags representing the status of the cache entry: is a disk write required, is a disk write in progress, is a disk load in progress, and so on.

This is not such an intimidating amount of data, but the correctness of the program can only be understood in the context of a system with an actual storage disk. For example, a desirable property of the cache is its *self-consistency*: If the client writes data to one of the pages, and then reads it later, it should observe the same data that was written. Of course, the page might be evicted from the cache in the meantime, and the data is only preserved through the storage disk. Therefore, reasoning about this property is only possible if we can reason about the behavior of the storage disk and the program's interaction with it.

This brings us to the first challenge:

Challenge SpC-1 (External devices). We need to be able to reason about the properties of a system where the program is but one component interacting with external devices.

Meanwhile, switching gears from high-level to low-level, **the locking mechanism** is what allows safe, multi-threaded access to the cache entries. To understand the locking mechanism, it is first important to understand the difference between the cache's internal locking scheme and its user-facing locking scheme. The key difference is that the client is primarily concerned with *disk pages*, indexed by their location on disk, while the cache internally has a lock *per cache entry*.

Here is a scenario that illustrates the difference between the two "levels" of locking:

- The client requests a read-lock for disk page d.
- The cache finds that disk page d is not in-memory, so it finds a free cache entry c and assigns it to d.
- Internally, the cache takes a *write-lock* on c so that it can it can safely load the contents of disk page d into c.
- When the load completes, the cache downgrades its lock to a read-lock.
- The cache returns to the client with the lock held.

In fact, the lock implementation has *specialized flows* that are optimized for these situations. When the cache takes a write-lock for the purposes of a disk load, as in the above scenario, it uses a slightly different procedure than it would when taking a write-lock due to a user request. Similarly, when the cache takes a read-lock for the purposes of performing a writeback, it again uses a different procedure than the one it would for a user read-lock. On top of all this, the lock has a special feature called *claiming* that allows a client to reserve the right to upgrade a read-lock to a write-lock.

The point is that this is *not* a lock that would be found in a general-purpose concurrency library. This brings us to our second challenge:

Challenge SpC-2 (Specialized Lock). We need to be able to reason about specialized lock implementations that support read-locks and write-locks. Read-locks may be taken simultaneously by multiple threads, while write-locks must be unique.

We have presented two challenges, and ordinarily, one might attempt to address these through modular levels of abstraction. For example, you could imaging building the lock as a library with a modular interface, and then use that to build the cache. However, SplinterCache is *not* implemented this way, and in fact, there is no way that it *could* be! It turns out that the lock is *inextricably* interlinked with higher-level cache logic.

For example, one of the flags in a cache entry's status field is the *writeback-in-progress* bit. This single field serves dual purposes. First, it is part of the cache metadata, used to keep track of outstanding asynchronous IO operations. However, it is *also* part of the lock: This bit represents a read-lock on the cache entry data, as part of the specialized writeback flow discussed above.

In other words, it isn't possible for the implementation to "factor out" the lock into its own module because parts of the lock *are* also important parts of the cache. This brings us to the final challenge for SplinterCache:

Challenge SpC-3 (Intertwining). Logic related to high-level cache domain logic is intertwined with low-level synchronization logic, which increases the complexity of the implementation.

2.4 Case Study II: Node Replication

Node Replication (NR) [7] is an algorithm designed to allow multi-threaded access to a data structure, optimized for high throughput on a NUMA architecture. In NUMA architecture—that is, architecture with *non-uniform memory access*—different processors have their most efficient memory access to different parts of memory. The core observation of the Node Replication algorithm is that we can improve memory locality by replicating the data structure across the NUMA nodes so that each node can efficiently access the copy of the data structure that is closest to it. The result is that the user can write their own *sequential* data structure, and NR automatically upgrades it to a concurrent, highly-parallelizable data structure. NR aims to present a linearizable view of the underlying data structure.

Node Replication is around ~ 1000 lines of Rust, some of which is "unsafe" Rust. It is used by NrOS [3], a research operating systems that uses NR to achieve high scalability with a relatively clean implementation. NrOS reports scaling up to 96 cores using Node Replication, and that it "nearly always dominates Linux at scale, in some cases by orders of magnitude."

Roughly, NR is organized as follows. The entire system is subdivided into nodes, and each node has multiple threads pinned to it. There is one copy of the data structure per node, which will only be accessed directly by threads from that node. To perform a *query* (a non-updating operation), a given thread will perform a query by accessing the data structure replica of its associated node. To perform an *update*, it must both modify its local data structure while also informing the other nodes about the operation so that they may apply the same operation to their own replicas. In order to coordinate and agree on a global ordering of update operations, all operations are communicated via a *global message buffer*.

2.4.1 Challenges

The first challenges that we encounter are reminiscent of those from from the previous section. First, each replica is stored in a reader-writer lock. NR uses its own implementation of a reader-writer lock, whose primary optimization is that it has multiple read-counters across different cache lines. This reader-writer lock is not as complicated or specialized as the one in SplinterCache, but nonetheless, it is a reader-writer lock that needs to be verified.

Notably, a similar but much more interesting situation occurs if we look at the global message buffer. In the global message buffer, each message written to it must be eventually read by each node so that its operation can be applied to each replica. Eventually, and only *after* it is read by each node, the message's slot may be reclaimed for another message. In this way, the message buffer acts "like a reader-writer lock" over each message slot:

- One thread may write to a slot;
- the slot is then read by many threads, possibly simultaneously;
- when they are done, a thread may write to the slot again;
- and so on.

However, despite this familiar-looking pattern, the message buffer does not resemble a traditional reader-writer lock in the slightest. Its operation is not dictated by a reference count, but by a series of index-pointers into the buffer.

The high-level point is similar to Challenge SpC-2, but it must now be stated to encompass this more general case.

Challenge NR-1 (Specialized lock-like system). We need to be able to reason about specialized implementations that support simultaneous read-states and exclusive write-states, *including those that do not resemble traditional reader-writer locks.*

Again like SplinterCache, NR exhibits the intertwining of high-level and low-level concerns (as in Challenge SpC-3). Recall that message buffer has several index-pointers that help ensure memory-safe access to the buffer, while an additional reader-writer lock helps ensure memory-

safe access to the replica. In fact, both of these things *also* play a crucial role in enforcing the high-level linearizability property. Specifically, they help ensure that queries are not performed on out-of-date replicas, the logic of which we will cover later.

In fact, NR originally had a bug related to this issue, which we identified over the course of developing our verified NR. Specifically, it turns out that the reader-writer locks, whose primary purpose is to provide safe access to the replicas, actually need to be held for the entire duration of the operations that access the message buffer, even though there is no safety-related reason to do so. In the original NR, there was a corner case where the lock would *not* be held as such, and as a result, we were able to identify a reproducible linearizability violation, where two different threads could observe reads out-of-order.

Challenge NR-2 (Intertwining). Logic related to high-level replication and linearizability domain logic is intertwined with low-level synchronization logic, which increases the complexity of the implementation.

Finally, there is one last challenge related to linearizability. Generally speaking, the easiest way to demonstrate linearizability is to identify the *linearization points*, the points that yield a total ordering over all operations. In NR, it happens that the exact position of the linearization points needed to make everything consistent might depend on nondeterministic decisions made *after* the locations of the linearization points. These are called *future-dependent linearization points*.

Challenge NR-3 (Future-dependent linearization points). We need to be able to prove linearizability even in the presence of future-dependent linearization points.

2.5 Case Study III: Mimalloc

Mimalloc [47] is a general-purpose userspace memory allocator—i.e., a drop-in replacement for `malloc` and `free`—originally designed for the Koka [45] and Lean [15] runtimes.

2.5.1 Challenges

Roughly speaking, the allocator works by dividing memory into *pages*, which are then divided into allocatable *blocks*. Each page arranges its available blocks into linked lists called *free lists*, where each available block contains a pointer to the next one.

The main challenge for memory allocator developers is that we cannot rely on our usual tools for managing memory, since many of those tools in turn rely on memory allocators. We have to allocate our memory directly from the OS, using a syscall like Linux's `mmap`. We need to manually manipulate the virtual address space, carving it up into memory for the allocator's internal data structures and memory to be allocated to the user.

Challenge Mem-1 (Fungible memory). We need to manually organize the address space, and safely divide the memory between internally-used memory and memory provided to the client used incorrectly.

In order to remain thread-safe, each individual thread maintains its own data structures from which it can allocate memory to the user. Unfortunately, this does not obviate all needs for multi-threaded considerations. The main issue is that the client may choose to call `free` on a chunk of memory that was allocated from a *different thread* than the one where it was originally allocated. In this case, the block needs to be returned to the data structure of its original thread. In order to do this safely, mimalloc has each thread maintain an *atomic free list*. Threads are able to safely insert blocks into this list via atomic compare-and-swap instructions. However, this whole process still involves "reaching into" the other threads' data structures in a carefully coordinated way.

To make everything worse, the correctness of all this somehow relies on the fact that `free` was called in the first place. What I mean is that the thread performing `free` has to reason something like this: "`free` was called for the pointer `p`, which means `p` is currently an existing un-freed allocation. Furthermore, `p` is in the *XY* range of memory, so the *XY* range must be a valid segment belonging to the allocator. It might belong to a different thread, but I can still make some conclusions about the validity of its state."

Challenge Mem-2 (Multi-threaded free). The correctness of `free` is reliant on the client calling it correctly, and from this information, we need to be able to make a number of deductions about the behavior of a multi-threaded system.

2.6 Case Study IV: Reference-Counted Smart Pointer

Our final case study is a little different than the last three. The implementation of a smart pointer is not particularly large or complicated, and our interest in it comes from its *ubiquitousness* rather than its scale. Even so, it is nontrivial.

The objective of a smart pointer is to manage a memory allocation by automatically freeing it after all handles to the allocation go out of scope. A *reference-counted* smart pointer does this by maintaining a "reference count" for each allocation that tracks the number of handles. Whenever a handle is destroyed, it checks if the count has reached 0, that is, it checks if there are no more handles. If so, it immediately performs a deallocation. Examples include C++'s `shared_ptr` or Rust's `Arc` or `Rc`.

2.6.1 Challenges

Traditionally, the main "point" of a reference-counted smart pointer is its convenience: The user of the smart pointer should not have to concern themselves with the pointer or the allocation at all. In a setting of formal reasoning, we want to maintain that convenience. Ideally, we would like to verify it in such a way that enables formal reasoning of the clients that *use* the smart

pointer. Specifically, we would like a client that uses a smart pointer, e.g., `Arc<T>` to be able to reason about it as if it were just a `T`.

Challenge RC-1 (Simple spec). The formal specification of a smart pointer should be easy and convenient to use by the client, matching the informal reasoning that a pointer-handle to a `T` is "like" a `T`.

Rust makes an interesting distinction with its two reference-counting types, `Arc` and `Rc`. Specifically, `Arc` is thread-safe, so that multiple threads may have a handle to the same allocation. Meanwhile, `Rc` is *not* thread-safe, so the handles for a particular allocation cannot be shared cross-thread, but as a result, it can be implemented in a simpler, faster way. (C++ does not make this distinction, incidentally; its `shared_ptr` is always thread-safe.)

Ideally, a formal methodology should be able to handle both kinds of implementations. Since the fast implementation is only correct when it is not used in a multi-threaded setting, it needs to **(i)** be possible to state this restriction in its user-visible spec and **(ii)** be possible to write a proof of correctness that somehow depends on this restriction.

Challenge RC-2 (Thread (non-)safety). Our framework should be able to handle objects that are *not* thread-safe and the more permissive implementations they permit, while still ensuring that they are not used incorrectly.

Finally, a common use-case for such types is in building recursive data structures, like binary trees, abstract syntax trees, or linked lists:

```
1 struct Tree {
2     Leaf,
3     InternalNode(Rc<Tree>, Rc<Tree>)
4 }
```

Challenge RC-3 (Recursive types). Our method needs to be consistent with the use-case of recursive data structures.

Case Study	(Desc., Soln.)	Challenges		Solved?
SplinterCache	(§2.3, §9.1)	Challenge SpC-1	External devices	✓
		Challenge SpC-2	Specialized lock	✓
		Challenge SpC-3	Intertwining	✓
Node Replication	(§2.4, §9.2)	Challenge NR-1	Specialized lock-like system	✓
		Challenge NR-2	Intertwining	✓
		Challenge NR-3	Future-dependent linearization points	✓
Mimalloc	(§2.5, §9.3)	Challenge Mem-1	Fungible memory	✓
		Challenge Mem-2	Multi-threaded free	✓
Rc / Arc	(§2.6, §9.4)	Challenge RC-1	Simple spec	✓
		Challenge RC-2	Thread (non-)safety	✓
		Challenge RC-3	Recursive types	✓
		Challenge WM	Weak memory ordering	✗

Table 2.1: **Summary of the challenges for the 4 case studies.**

2.7 Unaddressed challenges

It is only fair that we also describe the challenges we will be unable to handle. The memory allocator is a large case codebase, much larger than the other case studies, and our solution will only handle around 25% of it. There are some small technicalities for `Rc` and `Arc` that we won't be able to handle, which we'll discuss further in our solutions chapter, Chapter 9.

Finally, there is one challenge we are unable to address that I feel is significant enough to get a fancy box.

Challenge WM (Weak memory orderings). Efficient concurrent systems often take advantage of weaker memory orderings, such as "release/acquire" orderings or "relaxed" orderings.

This challenge is relevant to three of our case studies: the reference implementations for NR and mimalloc both use these memory orderings, and common implementations of thread-safe smart pointers often do as well. Unfortunately, our methodology only supports atomic operations that use *sequentially consistent* ordering (in addition to non-atomic, "ordinary" memory accesses). Thus, weaker orderings remain a challenge that we do not handle.

2.8 Recap

Table 2.1 summarizes the key challenges. Our next task will be to characterize a methodology capable of addressing these challenges. In Chapter 9, we will overview our solutions to these case studies, evaluate, and discuss the specific ways these challenges are addressed.

Chapter 3

Verus Overview

3.1 Two tools (but mostly one tool)

Motivated by challenges like the ones in the previous chapter, I have led the development of multiple tools exhibiting a common methodology to approach them. The first such tool is the **IronSync** framework [28], built in the verification language *Linear Dafny* that I also contributed to, the latter an extension of a popular verification language called *Dafny* [48]. Strictly speaking, IronSync refers to a specific framework in Linear Dafny that I built to handle complex concurrent systems code, but for the most part, I will use Linear Dafny and IronSync interchangably.

The second tool is **Verus**, a verification tool for Rust code. It may seem like verifying Rust code is a qualitatively different challenge than developing in a specialized verification language, but in fact, Verus is very much a continuation of Linear Dafny/IronSync.

I used these tools to approach the case studies. Working with other researchers, I led the verification of SplinterCache and NR implementations in IronSync. We later ported NR to Verus, and I also implemented the mimalloc case study in Verus. This is summarized in Table 3.1.

In this thesis, I hope to convey the key ideas of this common methodology—its essence, if you will—in a reusable way that transcends any particular tool. Nonetheless, I cannot write this thesis in a tool-agnostic way, especially since an important contribution of this work is the actual practical development of verified artifacts and their evaluations. And anyway, specificity is essential for the sake of exposition.

Furthermore, attempting to explain two different tools would only make this document bloated and unwieldy. **Therefore, this thesis will focus on Verus.** Not much will be much lost by focusing on Verus rather than Linear Dafny; Verus is essentially a continuation of Linear Dafny anyway, already incorporating most of what Linear Dafny did, and in most cases, improving on it. Also, Verus builds on Rust, a mainstream programming language, which makes it an easier vehicle for exposition, especially for the elements related to ownership.

We will come back to Linear Dafny eventually; I need to explain it so that we can properly compare Linear Dafny and Verus, and so that I can explain the SplinterCache implementation (though even there, I will "translate" Linear Dafny/IronSync into Verus so the reader can follow along without having to learn two different notations). In order to explain the core concepts, starting now and spanning the next several chapters, I will be focusing exclusively on Verus.

Case study	Programming language		Verification tool	
	C/C++	Rust	IronSync / Linear Dafny	Verus
SplinterCache	Reference-Cache [11]		IronSync-Cache	
NR		Reference-NR [7]	IronSync-NR	Verus-NR†
Mimalloc	Reference-Mimalloc [47]			Verus-Mimalloc
	Prior work (unverified)		Subject of thesis	

† Verus version with minor changes in design compared to IronSync version

Table 3.1: **Implementations of the major case studies across different tools.** *On the left half of the table are the original implementations of each one, which we call "reference implementations." These were all designed for production use, and were not designed with verification in mind. On the right half of the table are the versions that we built, aiming to match the original designs as closely as possible, in our verification frameworks.*

3.2 A Brief Overview of the Pieces

Briefly, our methodology consists of:

(1) A programming language, a specification language, a proof language, and an automated proof-checking engine.

(2) An ownership type system that makes verification easier by helping us avoid "surprising state modifications," as we discussed in the introductory chapter.

(3) A system of "ghost objects" operating within the ownership type system, allowing us to address challenges that would otherwise be impossible with more standard ownership types.

Most of the ghost object system is inspired by concurrent separation logic [61], and especially the *Iris* concurrent separation logic [35]. We take direct inspiration from the following aspects of CSL:

- *Memory permissions*, to handle data structures whose shapes make poor fits for traditional ownership types.
- *Ghost state* and *invariants*, to handle concurrent data structures and other situations where disparately owned objects need to maintain coordination.

(4) A novel "description language" for ghost state, which aids in concise specification of ghost operations and efficient verification conditions.

(5) A framework for handling interactions with external devices and other "whole system" properties. We delay talking about this one until Chapter 7.

That's the high-level picture. To expand on it, we will start with the basics of automated program verification, and then we will work our way up to the more unique aspects of our method.

3.3 Program Verification and Automation

3.3.1 Basics

Verus uses modular verification of functions based on preconditions and postconditions, discharging proof obligations using an SMT solver, similar to Dafny [48] and F* [70]. Let's unpack what this means.

First, *modular verification* means that verification is done considering one function (or lemma) at a time. The correctness of the whole program follows from the correctness of each individual function. The main advantage of this is that it keeps individual queries small, which both reduces pressure on the automated theorem prover and reduces mental load for the developer. A secondary advantage is that it makes verification highly parallelizable.

Next, the *preconditions* and *postconditions*, denoted by the `requires` and `ensures` clauses respectively, are used to indicate what is expected to be true before and after the execution of the function body.

```
1 fn test(a: u64, b: u64) -> (return_value: u64)
2     requires a <= 1000, b <= 900,
3     ensures return_value <= 1900,
4 {
5     return a + b;
6 }
```

The `requires` and `ensures` clauses are written in a logical specification language. Specifications are not executable, and in fact, they do not even need to be computable—they could use unbounded quantification, for example.

Preconditions are checked at call-sites, whereas postconditions are checked as part of the correctness of their functions. For example, every caller of `test` has to prove that its arguments satisfy `a <= 1000` and `b <= 900`. Meanwhile, when `test` itself is checked for correctness, we have to show that the postcondition holds subject to the precondition. This is done by computing the *weakest precondition* [19], a standard procedure for this kind of problem. In the case of our `test` function, the weakest precondition is fairly trivial; our verification condition becomes:

$$\forall a, b, r.\ (0 \leq a < 2^{64}) \wedge (0 \leq b < 2^{64}) \Rightarrow (0 \leq a + b < 2^{64} \wedge (r = a + b \Rightarrow r \leq 1900))$$

The first two conjuncts are implicit because `a` and `b` both have type `u64`, i.e., they are 64-bit unsigned integers. The proposition also demands that the result of the addition fit in a `u64` because part of Verus's job is to check that arithmetic operations do not overflow. Finally, the proposition demands that the return value (r) meet the requirement in the postcondition. This proposition can be proved with a trivial bit of linear arithmetic, which our theorem prover, Z3, solves easily.

Overall, this is a pretty simplistic example because the body of `test` is a single return statement. However, the weakest precondition is a well-established concept that can easily handle more complex bodies, including control flow and imperative statements that assign to variables.

3.3.2 Invariants, proofs, and lemmas

Unfortunately, most proofs are not so trivial. Let us look at a more complicated example, the Fibonacci example from the Verus paper [42], which both has more involved proofwork and which introduces us to several new concepts. See Figure 3.1

The function `fibo_impl` computes the n^{th} Fibonacci number for the input n. This is, in fact, exactly what the postcondition says (line 19). Observe that `fibo_impl` also has a precondition, namely that the n^{th} Fibonacci number *actually fits in a 64-bit integer*. Of course, we cannot hope to meet the postcondition if we called `fibo_impl` for a value of n where that wasn't true!

Also observe the way these two functions, `fibo` and `fibo_fits_u64` are defined: they are defined as *spec functions*. A spec function is exactly what it sounds like: a function to be used in specifications. These are not executable, and again, they do not need to be efficient or even computable. For example, the definition of `fibo` is recursive in an exponential way, but that doesn't matter since the function is not meant to be executed. Well-written spec functions are concise and mathematically precise; this one resembles the usual mathematical definition of the Fibonacci sequence, $F_{n+2} = F_{n+1} + F_n$. This is actually a great illustration of one of the chief values of verification: proving that an efficient implementation of a function matches a less efficient but more mathematically precise implementation.

Another small difference between the implementation and the specification are the integer types that are used. The implementation uses a common Rust integer type, `u64`, representing unsigned 64-bit integers. Bounded integer types are much more awkward for writing specification, so Verus provides `int` and `nat` types (only for use in specification), that represent $\mathbb{Z}$ and $\mathbb{N}$, respectively. Unbounded integer types make it easier to write conditions about overflow, as in line 14.

Now, let us turn our attention to the body of `fibo_impl`. This function is implemented with a while-loop, and to handle while-loops, Verus requires the user to supply a *loop invariant*. The loop invariant is shown here in the `invariant` clause (lines 28–33). Among others, one thing Verus needs to do is show that the invariant is *inductive*, i.e., that *if* it holds at the beginning of a loop iteration, then it also holds at the end of the end of that loop iteration.

There is one slight challenge with the loop body: Verus needs to show that the addition `cur + prev` (line 39) does not overflow. To do this, it needs to know that `fibo(i as nat)` fits in a `u64`. What's difficult about that? We already know that `i <= n`, and we also know (from the precondition of `fibo_impl`) that `fibo(n as nat)` fits in a `u64`. The issue is that what we really need is the fact that `i <= n ==> fibo(i as int) <= fibo(n as nat)`, but this fact is not obvious to the solver. To bridge this gap, we call a *lemma* (line 37) to introduce this fact.

This lemma is stated and proved at line 47. In Verus, lemmas are also called *proof functions*, denoted `proof fn`. The Verus solver handles this function the same way it would an executable function (i.e., by computing the weakest precondition). The lemma can be proved by induction, which here is represented by recursion. Since it is recursive, we also need to supply a *decreases-measure* (line 50) to prove that the recursion actually terminates.

```
spec fn fibo(n: nat) -> nat
    decreases n
{
    if n == 0 {
        0
    } else if n == 1 {
        1
    } else {
        fibo((n - 2) as nat) + fibo((n - 1) as nat)
    }
}

spec fn fibo_fits_u64(n: nat) -> bool {
    fibo(n) <= 0xffff_ffff_ffff_ffff
}

fn fibo_impl(n: u64) -> (result: u64)
    requires fibo_fits_u64(n as nat)
    ensures result == fibo(n as nat)
{
    if n == 0 {
        return 0;
    }
    let mut prev: u64 = 0;
    let mut cur: u64 = 1;
    let mut i: u64 = 1;
    while i < n
        invariant
            0 < i && i <= n,
            fibo_fits_u64(n as nat),
            fibo_fits_u64(i as nat),
            cur == fibo(i as nat),
            prev == fibo((i - 1) as nat),
    {
        i = i + 1;

        proof { lemma_fibo_is_monotonic(i as nat, n as nat); }

        let new_cur = cur + prev;
        prev = cur;
        cur = new_cur;
    }

    return cur;
}

proof fn lemma_fibo_is_monotonic(i: nat, j: nat)
    requires i <= j,
    ensures fibo(i) <= fibo(j),
    decreases j - i,
{
    if i < 2 && j < 2 {
        // No interesting proof work in this case
    } else if i == j {
        // No interesting proof work in this case
    } else if i == j - 1 {
        // Directive telling the solver how to expand fibo
        reveal_with_fuel(fibo, 2);
    } else {
        reveal_with_fuel(fibo, 2);
        lemma_fibo_is_monotonic(i, (j - 1) as nat);
    }
}
```

Figure 3.1: **Example of verified Fibonacci implementation.**

3.3.3 Proof automation

Core to the scalability of the proof method is the automation. So how exactly does it work? Just now, we saw an example where we had to write some extra proof code to help Verus make a deduction, walking it through an inductive proof step-by-step. But how does the solve these individual steps? What *can* it deduce automatically?

By default, Verus discharges all proof obligations via Z3 [16] using its theories of quantifiers, datatypes, uninterpreted functions, and linear arithmetic. In this mode, Z3 instantiates quantifiers through a mechanism called *trigger pattern matching*, wherein every quantifier is annotated with a trigger pattern that determines how it gets instantiated.

For example, consider a situation where some universally quantified proposition is in a hypothesis (or dually, we could consider an existentially quantified proposition in the conclusion):

$$\forall x.\ P(x) \Rightarrow Q(x)$$

This quantifier will have some trigger pattern associated with it. Usually (though not necessarily), the trigger pattern is a subexpression of the quantified predicate, so here it would probably be $P(x)$ or $Q(x)$. It might even be both. Let's suppose the trigger pattern is $P(x)$. Then if the solver finds an expression $P(e)$ in the context, it will instantiate the quantifier with $x = e$ to learn $P(e) \Rightarrow Q(e)$. Here, e could be a constant, a symbolic value, or any other expression.

Trigger pattern matching is just a heuristic, and it is, of course, incomplete. Its success also depends on the choice of the triggers. If there are too few triggers, then fewer proofs will be solved automatically, and the user may need to add extra assertions into the source code (thus adding more expressions into the context and causing more quantifiers to trigger). If there are too many triggers, the the solver's search space might become too big, leading to unresponsiveness and a poor feedback loop.

When Verus selects trigger sets, it deliberately errs on the side of minimal trigger sets. In many cases, the user is expected to supply the trigger annotations themselves, which can take some experience to get used to; one needs to develop an intuition for which expressions will be in the context. This unfortunately makes Verus a little less friendly than some other tools, but the choice was driven by our experience dealing with unresponsive solvers that resulted from aggressive trigger strategies. Verus also includes a profiler to help the user identify poorly chosen triggers that slow down the solver.

Verus also supports some additional prover modes for specialized situations, which the user can select on a case-by-case basis:

- Bitvector reasoning: Verus encodes all integers as bitvectors and uses Z3's bitvector theory.
- Nonlinear reasoning (Z3): Verus enables Z3's nonlinear arithmetic theory.
- Nonlinear reasoning (Singular): Verus sends the query to the Singular solver [17], which can solve many problems by computing a Gröbner basis. This is particularly useful for problems involving modular arithmetic.
- "By computation": Verus internally normalizes an expression. This is useful when a proof needs to unfold a definition hundreds or thousands of times.

Note that all of these solver modes constitute part of the trusted computing base.

3.3.4 Unsafe code and safety-relevant preconditions

In many cases, memory safety can be dependent on the preconditions of the function. As an illustrative example, we can take a look at Rust's vector-indexing functions.

In common vector-indexing functions (e.g., writing `vec[i]`), Rust always performs a *bounds check*. This means it is always memory-safe to call, since in the worst case, Rust will panic and exit the program early. However, there is a lesser-used `get_unchecked` operation which performs no such bounds-check. Thus `get_unchecked` is marked "unsafe," because the responsibility falls on the programmer to use it correctly, and if they fail, Rust may exhibit undefined behavior.

What does it mean to use the function "correctly"? Well, it means to call the function with an index that is actually in-bounds. In fact, we can represent this condition via a Verus precondition:

```
1 unsafe fn get_unchecked<T>(vec: &Vec<T>, idx: usize) -> &T
2     requires idx < vec.len()
3     ...
```

Rather than "unsafe," though, I like to call such functions "conditionally safe," i.e., they are safe to call *if* the preconditions hold. Of course, Verus *can* and *does* check the preconditions, making it much safer to use functions like `get_unchecked` in Verus-checked code.

One might wonder about more complex uses of "unsafe" code. The prototypical example of an unsafe Rust feature is a pointer access, but the "condition" needed to do that safely is incredibly complex. For one thing, it needs to be data-race-free, which is inherently a non-local program property. Can we really incorporate concepts like data-race-freedom into the framework of "conditional safety"? It turns out we can, though we will need to see more unique features of Verus to do it, such as the "ghost objects" we will introduce in §3.4.2.

3.4 An Overview of Verus's primitives

In this section, we will walk through the primitive types and features of Verus in detail. By "primitives," we mean that the features are not verified in terms of lower-level primitives; they are considered part of Verus's Trusted Computing Base (TCB). A summary of the primitives can be found in Table 3.2. As the table shows, many of the features are "native" to Rust, while the remainder are introduced by Verus. Most of what we cover in this section has to do with ownership in some way, so we will start by talking about Rust's ownership-related primitive types, and work our way up to Verus's unique primitive types.

It may be notable what is *not* in this table: many types taken for granted in Rust, such as `Rc`, `Arc`, `RefCell`, etc., are absent. This is because these types can actually be implemented and verified by composing the primitive types in this table.[1]

[1]Verus does actually include trusted specifications for the standard library's `Rc`, `Arc`, etc., but we don't use them for our case studies. The point we are illustrating in this thesis is that we *can* verify similar utilities, so they don't *need* to be trusted.

Feature	Ghost?	Typical Rust feature?
Primitives (`bool`, `u8`, ...)		✓
Tuples		✓
Slices `[T]`		✓
Arrays `[T; N]`		✓
Structs, enums, unions		✓
Marker traits (`Send`, `Sync`, `Copy`)		✓
References (`&` and `&mut`)		✓
`PCell`		Loosely based on `UnsafeCell`
`cell::PointsTo`	✓	
`PPtr`		Loosely based on `*mut T`
`ptr::PointsTo`	✓	
Atomics (`PAtomicBool`, ...)		Loosely based on `AtomicBool`, ...
`atomic::PermissionBool`, ...	✓	
Threading		✓
`spawn`, `join`		✓
`IsThread`	✓	
`AtomicInvariant`	✓	
`new`	✓	
`into_inner`	✓	
`open_atomic_invariant!`	✓	
`LocalInvariant`	✓	
`new`	✓	
`into_inner`	✓	
`open_local_invariant!`	✓	
User-defined ghost state	✓	
Ghost collections	✓	

Table 3.2: **Primitive types, functions, and traits in Verus.** *Types and functions colored in pink are "ghost."*

3.4.1 Ownership, references, and lifetimes

Rust's type system makes types *uniquely owned* by default. This means that it would be a type error to try to use an object after it is "moved" away:

```
fn example() {
    let mut object = Object::new();
    do_something(object);  // Move `object` away
    do_something_else(object); // Type error
}
```

In this small snippet, for example, you could imagine that `do_something` destroys `object`, maybe freeing resources or breaking some other kind of invariant, so that using `object` again later would be invalid. It is for this reason that Rust makes this code an error.

However, many types in Rust do not need such constraints. For example, basic primitives like `bool` or `u64` ought to be freely used as many times as we want. Such types are marked by a trait called `Copy`, which means they can be moved without being destroyed (i.e., you can "make copies" of them).

```
fn example() {
    let mut x: u64 = 24;
    do_something(x);
    do_something_else(x); // Completely fine
}
```

This system gets more interesting when we start talking about *references*. References are the most common pointer type in Rust, and they have interesting characteristics in the ownership type system, allowing the developer to temporarily "borrow" ownership of an object without moving it. Specifically, there are two reference types:

- The *immutable* reference, also called the *shared reference*, written `&T`.
- The *mutable* reference, also called the *unique reference*, written `&mut T`.

The first type, the shared borrow, is marked `Copy`, so the developer can freely make copies of the reference, hence an object can be referenced from multiple places; this is why it is called a *shared* borrow. The mutable borrow, however, is *not* marked `Copy`. Rust's type system enforces some key properties:

- You cannot have more than one active unique reference to the same object at the same time.
- You cannot have any active unique reference along with an active shared reference at the same time
- All borrows must "expire" (all references are dropped) before the original object can be moved. This property is accomplished through Rust's "lifetime" system.

Furthermore, unique references allow mutation of the underlying data, whereas shared references do not. This is often summed up through the catchphrase "aliasing XOR mutability," i.e., you can either have a reference that allows mutability (a single unique reference) or that allows aliasing (multiple shared references), but not both at the same time.

There is actually an exception to this, which we will come to later, but it is for this reason that I prefer the terms *shared reference* and *unique reference*. They are less ambiguous.

For now, let us accept "aliasing XOR mutability" at face value. It turns out that "aliasing XOR mutability" is the key property that lets us generate efficient verification conditions without having to directly deal with pointer indirection. Specifically, we can encode the reference to objects the same way we would as if we owned the object itself.

Why is this sound? The reason is that "aliasing XOR mutability" prevents unexpected mutations. Suppose, for instance, that we hold onto a shared reference. Then nobody is allowed to mutate it. Alternatively, suppose that we hold onto a unique reference. Then we might mutate it, but we are the *only* ones who can do so, so we can easily keep track of all its mutations.

To illustrate, I will show snippets of Rust/Verus code side-by-side with an "encoded" version. (Verus does not literally transform Rust code into Rust code; it uses an internal representation, but this will serve for illustrations' sake.)

Handling shared references is essentially trivial: we can ignore them completely.

```
fn example_shared_ref<'a, 'b>(x: &'a bool, y: &'b bool, z: &'b bool)
    -> Option<&'b bool>
{
    if (*x) && (*y) { Some(y) } else { Some(z) }
}

//// Encoded version:
fn encoded_example_shared_ref(x: bool, y: bool, z: bool) -> Option<bool> {
    if x && y { Some(y) } else { Some(z) }
}
```

We can handle unique references as function parameters by treating them as separate in-parameters and out-parameters:

```
fn example_negate(x: &mut bool) {
    *x = !(*x);
}
fn example_caller() {
    let mut t = true;
    example_negate(&mut t);
    assert(t == false);
}

//// Encoded versions:
fn encoded_example_negate(x_in: bool) -> (x_out: bool) {
    let x_out = !x_in;
    return x_out;
}
fn encoded_example_caller() {
    let mut t = true;
    t = encoded_example_negate(t);
    assert(t == false);
}
```

There is one significant limitation to Verus's current approach, though. Verus's support for **&mut** references is not fully general: Verus cannot verify functions that return **&mut** references, it

cannot handle structs with `&mut` references as fields, and it cannot instantiate generic parameters with a `&mut` type.

However, a solution to this does exist in the literature. RustHorn [55] has proposed an encoding scheme wherein `&mut` references are handled in full generality, still avoiding the need for explicit indirection-handling, by the use of a technique called "prophecy variables," and the encoding has found success in a different Rust verification tool, Creusot [18]. We intend to adopt this approach or something similar in Verus; it is currently in development. I will not be covering it in this thesis, however.

3.4.2 Interior Mutability and Cells

Rust supports a pattern called "interior mutability" wherein data can be modified even behind a shared `&` reference. This can only be done through special "interior mutability" types like `Cell`, `RefCell`, and `UnsafeCell`. Verus provides one general interior mutability type, which we call `PCell`, which together with other features can encompass many of the same use cases.

How is it possible for us to handle interior mutability in the first place? After all, we already established that "aliasing XOR mutability" is key to Verus's encoding, while interior mutability is the exception that breaks that rule.

Let us look at an example to see exactly what the problem is. Consider this code:

```
1  fn manipulate_2_cells(cell1: &Cell<u64>, cell2: &Cell<u64>) -> (u64, u64) {
2      cell1.set(24);
3      cell2.set(25);
4      let x = cell1.get();
5      let y = cell2.get();
6      (x, y)
7  }
8
9  fn example_cell() {
10     let cell = Cell::new(5);
11     let (x, y) = manipulate_2_cells(&cell, &cell);
12     // What are x and y?
13 }
```

The answer to the question in the comment (line 12) is that `x` and `y` will both be 25, since this is the value that was last written to the cell (line 3).

Now, what happens if Verus tried to represent `Cell<u64>` as a plain `u64`? Well, it would look something like this:

```
1  fn wrongly_encoded_manipulate_2_cells(cell1: u64, cell2: u64) -> (u64, u64) {
2      cell1 = 24;
3      cell2 = 25;
4      let x = cell1;
5      let y = cell2;
6      (x, y)
7  }
```

As a result, Verus would conclude that the function returns $(24, 25)$. Oops!

The central problem is that we assumed "aliasing XOR mutability" held when it did not, that is, we had shared references to the cells even though we could mutate the contents.

The key to implementing interior mutability soundly is to make sure that "aliasing XOR mutability" holds true *in the encoding*. To be more precise, the encoded representation of `Cell` or of any other cell-like object must be a value that does not change even when the "interior data" is modified.

This finally brings us to Verus's interior mutability type, `PCell`. The `PCell<V>` is represented in Verus's encoding only as an arbitrary identifier called a `CellID`. The encoding of a `PCell` does not contain the interior value at all. So then: how *do* we represent the interior value? The answer is to have an additional type, which we call `PointsTo<V>`. This is the first *ghost object* of interest to us, and its role is to map a `CellId` to an interior value. Thus, it is represented by a pair of values, `CellId` (given by `points_to.pcell()`) and the interior value `V` (given by `points_to.opt_value()`). Note that `opt_value()` returns an optional value—this lets us have the possibility of uninitialized cell data.[2]

Figure 3.2 shows the full `PCell` interface. (The keyword "tracked" here is an indication of ghost state that is ownership-tracked. In Verus proof code, the keyword `tracked` suffices; however, to be embedded in executable code or executable structs, such types must be wrapped in the `Tracked` type. When compiled, `Tracked` is equivalent to Rust's zero-sized `std::marker::PhantomData` type, though this is mostly just an implementation detail. For the sake of this thesis, we will not concern ourselves much with the difference between `tracked` and `Tracked`.)

Let us see this in action. First, let us rewrite `manipulate_2_cells` using `PCell`:

```
fn manipulate_2_pcells(
        cell1: &PCell<u64>, pt1: Tracked<&mut PointsTo<u64>>,
        cell2: &PCell<u64>, pt2: Tracked<&mut PointsTo<u64>>) -> (u64, u64)
    // This precondition says that:
    //   - `pt1` goes with cell `cell1`
    //   - `pt2` goes with cell `cell2`
    //   - both cells are uninitialized
    requires
        pt1.pcell() == cell1.id(), pt1.opt_value() == None,
        pt2.pcell() == cell2.id(), pt2.opt_value() == None,
{
    cell1.set(24, Tracked(&mut pt1));
    cell2.set(25, Tracked(&mut pt2));

    let x = cell1.get(Tracked(&pt1));
    let y = cell2.get(Tracked(&pt2));

    assert(x == 24);
    assert(x == 25);

    (x, y)
}
```

When *compiled*, the result will be similar to this (i.e., all ghost code disappears):

[2]It is important to note that we are not physically storing an option type. The `Option` only exists in the specification to represent the concept of 'initialized' versus 'uninitialized.'

```
fn manipulate_2_pcells(cell1: &UnsafeCell<u64>, cell2: &UnsafeCell<u64>)
    -> (u64, u64)
{
    unsafe {
        cell1.set(24);
        cell2.set(25);
        let x = cell1.get();
        let y = cell2.get();
        (x, y)
    }
}
```

But Verus's *encoding* is equivalent to something like this:

```
struct PointsTo_u64 {
    pcell: CellId,
    value: Option<u64>,
}

fn encoded_manipulate_2_pcells(
        cell1: CellId, pt1: PointsTo_u64,
        cell2: CellId, pt2: PointsTo_u64,
    ) -> (u64, u64, PointsTo_u64, PointsTo_u64)
    requires
        pt1.pcell == cell1,
        pt2.pcell == cell2,
        pt1.value == None,
        pt2.value == None,
{
    assert(pt1.pcell == cell1); // preconditions to cell1.set(...);
    assert(pt1.value == None);
    pt1.value = Some(24); // result of cell1.set(...);

    assert(pt2.pcell == cell2); // preconditions to cell2.set(...);
    assert(pt2.value == None);
    pt2.value = Some(25); // result of cell2.set(...);

    assert(pt1.pcell == cell1); // preconditions to cell1.get(...);
    assert(pt1.value.is_some());
    let x = pt1.value.unwrap();

    assert(pt2.pcell == cell2); // precondition to cell2.get(...);
    assert(pt2.value.is_some());
    let y = pt2.value.unwrap();

    assert(x == 24);
    assert(y == 25);

    (x, y, pt1, pt2)
}
```

Now, at this point, one may raise the following objection: **What's the point?** After all, our new function, `manipulate_2_pcells` forces the two input cells to be *different*, which is more restrictive than our original `manipulate_2_cells`.

```
struct PCell<V> { ... }
tracked struct PointsTo<V> { ... }
ghost type CellId = ...;

// Representations of PCell and PointsTo

impl<V> PointsTo<V> {
    pub spec fn id(&self) -> CellId;
    pub spec fn opt_value(&self) -> Option<V>;
}

impl<V> PCell<V> {
    pub spec fn id(&self) -> CellId;
}

// Primitive operations

impl<V> PCell<V> {
    pub fn new(v: V) -> (cell: PCell<V>, points_to: Tracked<PointsTo<V>>)
        ensures
            points_to.opt_value() == v && points_to.id() == cell.id();

    pub fn into_inner(self, Tracked(perm): Tracked<PointsTo<V>>) -> (out: V)
        requires
            self.id() == perm.id(),
            perm.opt_value().is_some(),
        ensures
            out == perm.value.unwrap();

    pub fn put(&self, Tracked(perm): Tracked<&mut PointsTo<V>>, v: V)
        requires
            old(perm).id() == self.id(),
            old(perm).opt_value() == None,
        ensures
            perm.id() == self.id(),
            perm.opt_value() == Some(v);

    pub fn take(&self, Tracked(perm): Tracked<&mut PointsTo<V>>) -> (out: V)
        requires
            old(perm).id() == self.id(),
            old(perm).opt_value().is_some(),
        ensures
            perm.id() == self.id(),
            perm.opt_value() == None,
            out == old(perm)@.value.unwrap();

    pub fn borrow<'a>(&'a self, Tracked(perm): Tracked<&'a PointsTo<V>>)
        -> (out: &'a V)
        requires
            perm.id() == self.id(),
            perm.opt_value().is_some(),
        ensures
            *v == perm.opt_value().unwrap();
}
```

Figure 3.2: **The `PCell` interface.**

It appears that, by forcing the user to have unique ownership of the PointsTo in order to modify the cell, we have entirely defeated the point of interior mutability. After all, interior mutability is *supposed* to be an escape hatch from the ownership restrictions, but we have merely shunted the main ownership restrictions from PCell to the PointsTo.

The short answer to this is that Verus has additional features for manipulating ghost objects like PointsTo that do not make sense for physical objects like cells. These special ghost features allow us to "coordinate" access to the cell in nontrivial ways. We will discuss some of these ghost features shortly. For the time being, though, we are not quite done with all the physical types.

3.4.3 Heap pointers

How do we handle heap allocations and pointers into the heap? We can handle this roughly the same way as cells, splitting into two types, a pointer type PPtr and a PointsTo type providing a permission and a value. In fact, this is a more traditional use of the "points-to" concept than the cell's PointsTo. Since there are now two "points-to" types, we will use cell::PointsTo and ptr::PointsTo to disambiguate them when necessary.

The PPtr API is shown in Figure 3.3. We can see that this is quite similar to the PCell interface (Figure 3.2).

The main difference is that a pointer does not actually contain the data, so it can be marked Copy, whereas the PCell is shared by reference. A heap allocation is always at a fixed place, whereas a PCell could potentially move around. This contrast can be seen in borrow, for example. In PCell::borrow, the lifetime of the resulting borrow is bound by the lifetimes of *both* the &PCell and the &PointsTo. This is because the data needs to remain at a fixed location for the resulting borrow to be valid. In PPtr::borrow, however, there is no lifetime on the pointer; the location of the allocation is always fixed. Another small difference is the presence of Dealloc token needed to call free().

Though I'm not presenting the spec for it here, there is also a "fungible" version of PointsTo that represents the permission for an arbitrary range of memory, called PointsToRaw. Given a PointsToRaw for a range of memory, with the appropriate size and alignment for an object of type V, we can convert to and from PointsTo<V>. The existence of this extended API is one reason we need the Dealloc token.

Note that the pointer API only supports heap pointers. Currently, there is no way to do anything meaningful with a stack variable, to a field in a struct, or into a cell.

3.4.4 Atomics

It is, of course, crucial to support atomic operations for the sake of multi-threaded algorithms. Rust's standard library provides atomic types for many different primitives (e.g., AtomicU64, AtomicBool, etc.) supporting operations like atomic fetch_or, fetch_add, and compare_exchange across a range of "ordering levels": SeqCst (sequentially consistent), Release, Acquire, and Relaxed. These atomics are considered to be interior mutability types.

Verus contains variants of these types that use ghost state permissions, in much the same way that PCell does. Following the naming convention, they are called PAtomicBool, PAtomicU64,

```
struct PPtr<V> { ... }
tracked struct PointsTo<V> { ... }
ghost type Id = ...;

impl<V> Copy for PPtr<V> { }

// Representations of PPtr, PointsTo, and Dealloc
impl<V> PointsTo<V> {
    pub spec fn pptr(&self) -> Id;
    pub spec fn opt_value(&self) -> Option<V>;
}
impl<V> Dealloc<V> {
    pub spec fn pptr(&self) -> Id;
}
impl<V> PPtr<V> {
    pub spec fn id(&self) -> Id;
}

// Primitive operations
impl<V> PPtr<V> {
    pub fn alloc(v: V) -> (ptr: PPtr<V>, Tracked(perm): Tracked<PointsTo<V>>,
        Tracked(dealloc_perm): Tracked<Dealloc<V>>
    )
        ensures
            perm.opt_value() == v,
            perm.pptr() == ptr.id(),
            dealloc_perm.pptr() == ptr.id(),

    pub fn free(self, Tracked(perm): Tracked<PointsTo<V>>,
        Tracked(dealloc_perm): Tracked<Dealloc<V>>
    ) -> (out: V)
        requires
            self.id() == perm.pptr(),
            self.id() == dealloc_perm.pptr(),
            perm.opt_value().is_some(),
        ensures
            out == perm.value.unwrap();

    pub fn put(self, Tracked(perm): Tracked<&mut PointsTo<V>>, v: V)
        requires
            old(perm).pptr() == self.id(),
            old(perm).opt_value() == None,
        ensures
            perm.pptr() == self.id(),
            perm.opt_value() == Some(v);

    pub fn take(self, Tracked(perm): Tracked<&mut PointsTo<V>>) -> (out: V)
        requires
            old(perm).pptr() == self.id(),
            old(perm).opt_value().is_some(),
        ensures
            perm.pptr() == self.id(),
            perm.opt_value() == None,
            out == old(perm)@.value.unwrap();

    pub fn borrow<'a>(self, Tracked(perm): Tracked<&'a PointsTo<V>>) -> (out: &'a V)
        requires
            perm.pptr() == self.id(),
            perm.opt_value().is_some(),
        ensures
            *v == perm.opt_value().unwrap();
}
```

Figure 3.3: **The `PPtr` interface.**

and so on. Verus's types only support the `SeqCst` memory ordering. `SeqCst` is the slowest memory ordering, but it is also the easiest to reason about. In particular, using `SeqCst` allows us to apply a concurrency reasoning principle: DRF+SC=SC, i.e., "Data-race-free plus sequentially consistent is sequentially consistent." Essentially, this principle says that for any program, if for any SC execution of that program, any non-SC assess is data-race-free, then any valid execution is SC. This allows us to reduce reasoning about concurrent programs to reasoning about SC programs.

Doing nontrivial things with atomics requires some additional concepts, so we will discuss those first, and then return to atomics in §3.5.4.

3.4.5 Ghost Invariants

Now that we have established ghost objects, we need more ways to manipulate them. Here we introduce two: `AtomicInvariant` and `LocalInvariant`. Here, the word "invariant" is taken from separation logic, where an invariant can be thought of as a kind of "container" that stores a proposition. The container can be shared, and it permits clients to temporarily obtain ownership of the contents.

In Verus's case, the thing stored in the invariant "container" is a ghost object. Specifically, an `AtomicInvariant<_, T, _>` or `LocalInvariant<_, T, _>` stores a ghost object of type `T`. (We will talk about the other type arguments below.) The key consideration for these objects is: how can we ensure that it is sound for the clients to obtain ownership of the contained ghost object? After all, there may be multiple references to any given Invariant object, so we must be careful that we do not accidentally give ownership of an object to different clients at once.[3]

First, let us consider the problem of splitting ownership across threads. There are two different ways to approach this, hence the two different types:

- `LocalInvariant` may not be used in a cross-thread way at all (it is "local" to a thread).
- `AtomicInvariant` may be used cross-thread, but it may not be opened for longer than the duration of a single atomic step.

For the purposes of the second check, certain primitive operations are marked as "atomic," specifically the ones associated with the atomic types (§3.4.4). Note that "ordinary" memory operations are *not* marked as atomic because these operations are required to be data-race-free.

Even within a single thread, there are some issues we need to be careful about. We need to disallow code like the following:

```
pub fn open_invariant_twice(Tracked(inv): Tracked<&LocalInvariant<T>>) {
    open_local_invariant!(inv, inner_obj1 => {
        open_local_invariant!(inv, inner_obj2 => {
            // Here with `inner_obj1` and `inner_obj2`, we have double-ownership
            // of the same object.
        });
    });
}
```

[3]There is one other important soundness consideration that is well-known in the higher-order separation logic community; we discuss this further in §6.5.

In order to properly disallow this, Verus tracks the invariants that are open at any given time, so it will always give an error in this situation. Of course, Verus needs to detect problematic situations across function boundaries as well, so to support modular verification, each function declares in its signature the invariants it might open.

The following snippets illustrate some of the mistakes a developer might make and the errors Verus provides:

```
pub fn f1(Tracked(inv): Tracked<&LocalInvariant<T>>)
    opens_invariants none,
{
    // Verus reports an error for this because it tries to open an invariant,
    // although the function signature says no invariants should be opened.
    open_local_invariant!(inv, inner_obj1 => {
    });
}
```

```
pub fn f2(Tracked(inv): Tracked<&LocalInvariant<T>>)
    opens_invariants any,
{
    open_local_invariant!(inv, inner_obj1 => {
    });
}

pub fn f3(Tracked(inv): Tracked<&LocalInvariant<T>>)
    opens_invariants any,
{
    open_local_invariant!(inv, inner_obj1 => {
        // Verus reports an error because `f2` specifies that it may open any
        // invariant, which might (and in this case, does) conflict with the
        // already-opened invariant.
        f2(Tracked(inv));
    });
}
```

Luckily, this system does not impose much burden on the verification condition generation, as opening an invariant is a somewhat rare operation, relatively speaking. If the user does not supply any explicit signatures relating to invariants, and if they never open any invariants, then there are no nontrivial invariant-related proof obligations.

Despite this tracking, it is still possible to open multiple invariants at once, as long the user proves that they are distinct. Based on a mechanism used by the Iris separation logic, Verus is able to distinguish different invariant objects from each other via their *namespaces*.

```
pub fn example_two_invs(
    Tracked(inv1): Tracked<&LocalInvariant<T>>,
    Tracked(inv2): Tracked<&LocalInvariant<T>>
)
    requires inv1.namespace() != inv2.namespace(),
    opens_invariants any,
{
    open_local_invariant!(inv1, inner_obj1 => {
        open_local_invariant!(inv2, inner_obj2 => {
            // This is fine; since `inv1` and `inv2` are different invariant
            // objects, both `inner_obj1` and `inner_obj2` are distinct objects,
            // and therefore, we don't have any kind of ill-formed
            // double-ownership.
        });
    });
}
```

The user can choose a namespace upon the creation of an invariant. (Note that there is no requirement for invariants to have unique namespaces, nor even a way of enforcing such a thing. This is perfectly sound, since Verus only needs a way to tell invariants apart, not the other way around.)

One last thing we have to handle here is the specification of an *invariant predicate*. The idea is that we usually want to constrain the values we store in the invariant container in some way. For example, if we are storing a `PointsTo` in the container, we might want to specify which location (pointer or `CellId`) it is associated with.

To do this, we can specify the invariant predicate via a trait:

```
pub trait InvariantPredicate<C, V> {
    spec fn inv(c: C, v: V) -> bool;
}
```

Now it is time to reveal the remaining type parameters of the invariant types:

```
type AtomicInvariant<C, V, Pred: InvariantPredicate<C, V>>
type LocalInvariant<C, V, Pred: InvariantPredicate<C, V>>
```

The `Pred` type parameter is used to specify the invariant predicate. The invariant predicate is a boolean predicate defined over `C` and `V`, where `C` is the type of a constant that can be configured upon creation of the invariant. See Figure 3.4.

As an example, Figure 3.5 shows how to use `LocalInvariant` to store an invariant that a cell contains an even integer. We use `C` to fix the `CellId`; note that this lets us talk about `CellId` of `LocalInvariant` without having to open it (e.g., on line 19). In the predicate specified by `Pred`, we tie to the `CellId` of the `PointsTo` to the invariant's constant value (line 7) and also constrain the `PointsTo`'s value to be even.

3.4.6 Thread-safety and the Send and Sync marker traits

Above, we stated that `LocalInvariant` cannot be used in a cross-thread manner. How do we enforce that? In fact, Rust already has a mechanism that is perfectly suited for this. Rust is able to track the thread-safety properties of a type via the `Send` and `Sync` marker traits.

```
1  pub trait InvariantPredicate<C, V> {
2      spec fn inv(c: C, v: V) -> bool;
3  }
4
5  tracked type AtomicInvariant<C, V, Pred: InvariantPredicate<C, V>>;
6  tracked type LocalInvariant<C, V, Pred: InvariantPredicate<C, V>>;
7
8  ghost type Namespace = int;
9
10 impl<C, V, Pred: InvariantPredicate<C, V>> LocalInvariant<C, V, Pred> {
11     pub spec fn constant(self) -> C;
12     pub spec fn namespace(self) -> Namespace;
13
14     pub proof fn new(c: C, tracked v: V, ns: Namespace) -> (tracked inv: Self)
15         requires
16             Pred::inv(c, v),
17         ensures
18             inv.constant() == c,
19             inv.namespace() == ns;
20
21     pub proof fn into_inner(tracked self) -> (tracked v: V)
22         ensures Pred::inv(self.constant(), v),
23         opens_invariants [ self.namespace() ]
24 }
25
26 impl<C, V, Pred: InvariantPredicate<C, V>> AtomicInvariant<C, V, Pred> {
27     // Identical to lines 11-23
28 }
29
30 // Opening invariants
31 open_local_invariant!(...);
32 open_atomic_invariant!(...);
```

Figure 3.4: **The `AtomicInvariant` and `LocalInvariant` interfaces.**

```rust
// We need a dummy type to implement the trait on.
ghost struct EvenCellPred { }

impl InvariantPredicate<CellId, PointsTo<u8>> for EvenCellPred {
    open spec fn inv(cell_id: CellId, points_to: PointsTo<u8>) -> bool {
        // The `points_to` proposition correctly corresponds to the cell
        points_to.id() == cell_id
        // And it points to an initialized, even integer value.
          && (match points_to.opt_value() {
              None => false,
              Some(x) => x % 2 == 0,
          })
    }
}

fn add_2(cell: &PCell<u8>,
    Tracked(inv): Tracked<&LocalInvariant<CellId, PointsTo<u8>, EvenCellPred>>
)
    requires inv.constant() == cell.id(),
{
    open_local_invariant!(inv => points_to => {
        // Upon opening the invariant, we gain access to the `points_to`
        // object, and we learn that it satisfies the given invariant.
        assert(points_to.is_init());
        assert(points_to.value() % 2 == 0);

        // Using the `points_to` we can perform cell operations.
        // Here, we add 2 to the given value, wrapping if necessary.
        let x = cell.take(Tracked(&mut points_to));
        assert(x % 2 == 0);

        // Add 2 (wrap around if necessary)
        let x_plus_2 = if x == 254 { 0 } else { x + 2 };

        cell.put(Tracked(&mut points_to), x_plus_2);

        // In order to close the invariant, we need to ensure that
        // the invariant predicate holds again. Our SMT solver
        // can prove this through trivial linear arithmetic.
        assert(points_to.is_init());
        assert(points_to.value() % 2 == 0);
    });
}

fn main() {
    let (cell, Tracked(points_to)) = PCell::new(4);

    let tracked inv = LocalInvariant::new(
        cell.id(),
        points_to,
        1337 /* arbitrary namespace */);

    add_2(&cell, Tracked(&inv));
    add_2(&cell, Tracked(&inv));
    add_2(&cell, Tracked(&inv));
}
```

Figure 3.5: **Example illustrating the use of an `InvariantPredicate`.**

```
impl<V> Send for PCell<V>
impl<V> Sync for PCell<V>

impl<V: Send> Send for cell::PointsTo<V>
impl<V: Sync> Sync for cell::PointsTo<V>

impl<V> Send for PPtr<V>
impl<V> Sync for PPtr<V>

impl<V: Send> Send for pptr::PointsTo<V>
impl<V: Sync> Sync for pptr::PointsTo<V>

impl<C, V: Send, P> Send for LocalInvariant<C, V, P>

impl<C, V: Send, P> Send for AtomicInvariant<C, V, P>
impl<C, V: Send, P> Sync for AtomicInvariant<C, V, P>
```

Figure 3.6: **Send and Sync marker traits for primitive Verus types.**

In short, if a type is Send, then ownership of the item may be transferred to another thread. If a type is Sync, then shared references to the type may be transferred to another thread. Figure 3.6 shows the Send/Sync traits for the types we have discussed so far.

Observe that PCell and PPtr are *always* marked Send and Sync; after all, a pointer is just an address, and a PCell is just bytes. Without the respective PointsTo object, it is not possible to retrieve the underlying V or &V.[4]

Also observe the crucial distinction between the two kinds of ghost invariant types. Specifically, LocalInvariant is never Sync (as this would erroneously allow the user to acquire double-ownership over the contained object across two different threads) while AtomicInvariant *is* Sync (assuming V is).

The marker traits also give a convenient way to talk about "thread IDs." We can use a ghost object called IsThread to represent the current thread ID. This is only sound because we are able to restrict IsThread from being sent between threads.

3.4.7 User-defined ghost state

The last element is user-defined ghost state. This is the most complicated element to address, and in fact, we will spend several chapters on it.

The short short story is Verus allows users to define ghost state using a novel "ghost state description language" that we call VerusSync. In VerusSync, the user defines the ghost state they want, the transitions they want to perform with it, and prove invariants and other well-formedness conditions on it. In exchange, VerusSync creates an API of "ghost tokens" that are useful for verifying concurrent code.

We will characterize VerusSync more formally in Chapter 5. In this chapter, we will present an example-driven introduction to VerusSync, starting in §3.6.

[4] I often describe PCell as "based on Rust's UnsafeCell," but this is actually one way in which it differs. The marker traits of an UnsafeCell are far less permissive.

3.5 Examples

Now, we will go through a series of bite-sized examples to show how the above-mentioned primitives can be combined to solve nontrivial verification problems. The main goal of this section is to provide more intuition about the pieces and what they are useful for.

3.5.1 Doubly-linked list

Earlier, we used the doubly-linked list as an example of an elementary data structure that can be difficult to do in Rust. Here, we will see how Verus's memory permissions let us tackle this problem.

First, let us define the `Node` type. Each node should have a `prev` pointer and a `next` pointer. Each pointer will be optional (for the first node of the list will have its `prev` pointer set to `None`; likewise, the last node of the list will have its `next` pointer set to `None`). Also, each node will store some value, `V`.

```
struct Node<V> {
    prev: Option<PPtr<Node<V>>>,
    next: Option<PPtr<Node<V>>>,
    value: V,
}
```

Recall that in the `PPtr` interface (§3.4.3), we need to use the ghost `PointsTo` objects to access the pointers. Thus, we need to answer: how should we manage this ghost state?

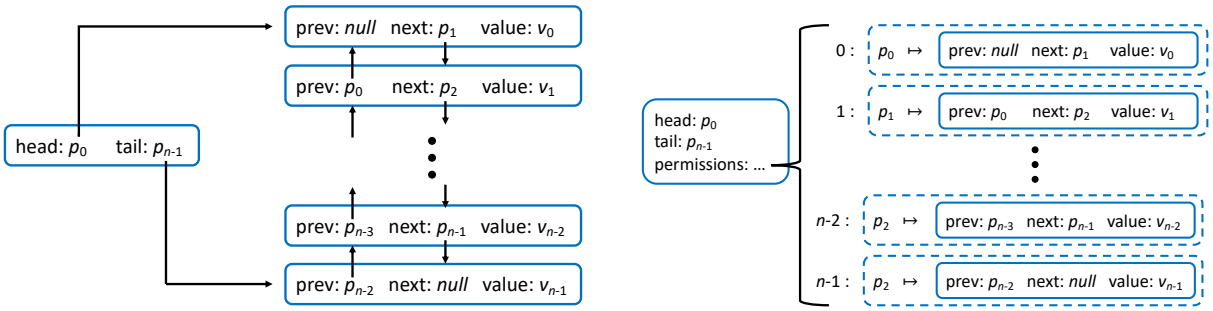
Ordinarily, when (say) node p is a parent of node n, we put the permissions to access n in p. However, the whole point of a doubly-linked list is that no node has a single parent. Each node is pointed to by two nodes, and crucially, it should be possible to reach any node by traversing from either direction.

In order to make sure all `PointsTo` objects are always accessible, we keep them in a flattened structure at the top level.

```
type MemPerms<V> = (PointsTo<Node<V>>, Dealloc<Node<V>>);

pub struct DoublyLinkedList<V> {
    ptrs: Ghost<Seq<PPtr<Node<V>>>>,
    perms: Tracked<Map<nat, MemPerms<V>>>,
    head: Option<PPtr<Node<V>>>,
    tail: Option<PPtr<Node<V>>>,
}
```

Physically speaking, a `DoublyLinkedList<V>` consists of two pointers, a head and a tail pointer, respectively. (Again, these are optional because the list may be empty, i.e., they might have nothing to point to.) We also keep a ghost `Map` of all the `PointsTo` objects, indexed from 0 to $n - 1$ where n is the length of the list at any time. Finally, we keep a ghost sequence of all the pointers (though this is technically redundant, and it is mostly for convenience). Figure 3.7 graphically illustrates the difference between the "physical structure" of the doubly-linked list and its "ghost ownership structure."



(a) Physical pointer structure of a doubly-linked list.

(b) Ownership structure of a Verus doubly-linked list, which includes ghost state.

Figure 3.7: **Doubly-linked lists.** *The dashed boxes are ghost* `PointsTo` *objects. Figure is from the Verus paper [42].*

Observe now that any client who owns the `DoublyLinkedList<V>` object can always access all the `PointsTo` objects. To traverse forward, they would start at 0 and move forward, while to traverse backward, they would start at $n - 1$ and move backwards.

Figure 3.8 formally shows the well-formedness predicate for a `DoublyLinkedList<V>`. At a high level, it simply says that all the data (both ghost and physical) in the `DoublyLinkedList<V>` struct is self-consistent, i.e., the `PointsTo` objects have the correct pointers; the nodes they point to have the correct `prev` and `next` pointers, and so on.

I will not go through every operation of a doubly-linked list here; Figure 3.9 illustrates a single operation, **`self`**`.push_back(v)`, and its verified implementation. This shows how we manipulate the ghost state alongside the physical state while preserving the doubly-linked list's invariants.

```
impl<V> DoublyLinkedList<V> {
    // Pointer to the node before index `i` (or None if there is none)
    spec fn prev_of(&self, i: nat) -> Option<PPtr<Node<V>>> {
        if i == 0 {
            None
        } else {
            Some(self.ptrs@[i as int - 1])
        }
    }

    // Pointer to the node after index `i` (or None if there is none)
    spec fn next_of(&self, i: nat) -> Option<PPtr<Node<V>>> {
        if i + 1 == self.ptrs@.len() {
            None
        } else {
            Some(self.ptrs@[i as int + 1])
        }
    }

    // Predicate indicating that the permission at index i is well-formed.
    spec fn wf_perm(&self, i: nat) -> bool {
        // i must be in the permission map
        self.perms@.dom().contains(i)
        // the permission objects must match the i^th pointer
        && self.perms@[i].0.pptr() == self.ptrs@[i as int].id()
        && self.perms@[i].1.pptr() == self.ptrs@[i as int].id()
        // the i^th node has the correct `prev` and `next` values
        && match self.perms@[i].0@.value {
            Some(node) => node.prev == self.prev_of(i)
                       && node.next == self.next_of(i),
            None => false,
        }
    }

    // Main well-formedness predicate for a DoublyLinkedList
    pub closed spec fn well_formed(&self) -> bool {
        forall|i: nat| 0 <= i && i < self.ptrs@.len() ==> self.wf_perm(i)
        && (
            if self.ptrs@.len() == 0 {
                self.head.is_none() && self.tail.is_none()
            } else {
                self.head == Some(self.ptrs@[0])
                  && self.tail == Some(self.ptrs@[self.ptrs@.len() as int - 1])
            }
        )
    }

    // Abstract representation of a doubly-linked list as a sequence of values.
    pub closed spec fn view(&self) -> Seq<V> {
        Seq::<V>::new(self.ptrs@.len(),
            |i: int| self.perms@[i as nat].0.value().value)
    }
}
```

Figure 3.8: **Well-formedness predicate and view for a `DoublyLinkedList`.**

```rust
/// Insert a single element into a list, assuming it is empty
fn push_empty_case(&mut self, v: V)
    requires old(self).well_formed(), old(self).ptrs@.len() == 0,
    ensures self.well_formed(), self@ == old(self)@.push(v),
{
    let (ptr, Tracked(points_to), Tracked(dealloc)) = PPtr::new(
        Node::<V> { prev: None, next: None, value: v });
    proof {
        self.ptrs@ = self.ptrs@.push(ptr);
        self.perms.borrow_mut()
            .tracked_insert((self.ptrs@.len() - 1) as nat,(points_to, dealloc));
    }
    self.tail = Some(ptr.clone());
    self.head = Some(ptr);
}

/// Insert a single element to the back of the list
pub fn push_back(&mut self, v: V)
    requires old(self).well_formed(),
    ensures self.well_formed(), self@ == old(self)@.push(v),
{
    match &self.tail {
        None => {
            // Special case: list is empty
            self.push_empty_case(v);
        },
        Some(tail_ptr) => {
            // Get the PointsTo for the last node in the list.
            // Remove it from the collection so we can manipulate it.
            let tracked (mut tail_perm, tail_dealloc) = self.perms.borrow_mut()
                .tracked_remove((self.ptrs@.len() - 1) as nat);
            // Read the tail node from the pointer.
            let mut tail_node = tail_ptr.take(Tracked(&mut tail_perm));
            // Allocate a new node, which will be the new tail node.
            let (ptr, Tracked(points_to), Tracked(dealloc)) = PPtr::new(
                Node::<V> { prev: Some(tail_ptr.clone()), next: None, value: v },
            );
            // Change the old tail node to point to the new tail node, and write it
            // back via the pointer.
            tail_node.next = Some(ptr.clone());
            tail_ptr.put(Tracked(&mut tail_perm), tail_node);
            proof {
                // Put the PointsTo for the previous tail node (now the
                // second-to-last node) back into the collection.
                self.perms.borrow_mut().tracked_insert(
                    (self.ptrs@.len() - 1) as nat,
                    (tail_perm, tail_dealloc),
                );
                // Put the PointsTo for the *new* tail node into the collection.
                self.perms.borrow_mut()
                    .tracked_insert(self.ptrs@.len(), (points_to, dealloc));
                // Update the pointers list.
                self.ptrs@ = self.ptrs@.push(ptr);
            }
            // Set `self.tail` to the new tail node.
            self.tail = Some(ptr);
        },
    }
}
```

Figure 3.9: **Verified implementation of `DoublyLinkedList::push_back`.** *To reduce clutter, code is simplified to remove some assertions that are necessary to help Z3 validate the proof. The focus of this presentation is on the manipulation of tracked ghost objects.*

3.5.2 More Cells

Earlier, we introduced our interior mutability primitive, `PCell`, which requires additional ghost object (`PointsTo`) to use. However, this is often inconvenient. By contrast, vanilla Rust provides a variety of safe interior mutability primitives that require no such thing, e.g., `Cell`.

In this section, we will see how we can implement and verify a more convenient `Cell`-like type. The key idea is to bundle the `PointsTo` object with the `PCell`. Of course, for this idea to work, we need to be able to obtain write-access to the `PointsTo` even when we have a shared reference our `Cell`-like object. This is exactly the situation that a `LocalInvariant` is suitable for:

```
1  pub struct InternalCellPred {}
2
3  impl<T> InvariantPredicate<CellId, PointsTo<T>> for InternalCellPred {
4      spec fn inv(cell_id: CellId, points_to: PointsTo<T>) -> bool {
5          points_to.pcell() == cell_id
6            && points_to.opt_value().is_some()
7      }
8  }
9
10 pub struct Cell<T> {
11     pcell: PCell<T>,
12     perm_inv: Tracked<LocalInvariant<CellId, PointsTo<T>, InternalCellPred>>,
13 }
14
15 impl<T> Cell<T> {
16     pub closed spec fn wf(&self) -> bool {
17         self.perm_inv@.constant() == self.pcell.id()
18     }
19 }
```

Here, our `LocalInvariant<...>` contains a `PointsTo<T>` and is parameterized by a `CellId`. The invariant (lines 4–7) ensures that the `PointsTo<T>` points to the correct `CellId`, while the Cell's well-formedness condition (lines 16–18) ensures that the `LocalInvariant`'s `CellId` matches that of the actual cell.

In order to read or write to the cell, we simply open the invariant to obtain the `PointsTo`, perform the desired operation, and close the invariant.

Of course, it would be even better if our `Cell` type allowed the client to specify a predicate to determine the allowed values that can be stored in the cell. Figure 3.10 presents an expanded version of `Cell` with this property, called `InvCell`.

In the `InvCell` interface, we have a trait, `InvCellPredicate`, that allows the client to specify a predicate on allowed values of `T`. To specify the predicate, they provide an implementation of the trait `P: InvCellPredicate`. Observe that `InternalInvCellPred`, the predicate defined for the `LocalInvariant`, is defined in terms of the client's invariant, `P::inv` (line 11).

Let us look at `replace` (lines 36–48). In terms of executable code, the one important line is line 45. which inserts `new_value` into the `PCell` and extracts the `old_value` to return to the user. To perform this operation, we have to obtain the `PointsTo` from the `LocalInvariant`. When we obtain the `PointsTo`, we know that it satisfies the predicate (line 44). This is why we are able to satisfy the postcondition (line 41). Likewise, when we close the invariant-block, we need to show that the newly-updated value of `points_to` also satisfies the invariant; this

follows because of the precondition (line 39).

3.5.3 Example: Using cells to memoize an expensive computation

An easy example that illustrates the utility of `InvCell` is that of memoizing an expensive computation (Figure 3.11).

For simplicity, our example assumes that the computation of interest is a 0-argument function (even though this obviates the utility somewhat). An executable function that computes the result is given by `expensive_computation`, while `result_of_computation()` is a spec version. Observe that the more efficient function, `memoized_computation`, has a postcondition (line 29) that says it returns the same value—that is, `result_of_computation()`—but the postcondition reveals no other information about *how* it is computed.

The actual way it is computed is by using a cell to store the result. The trait impl of `InvCellPredicate` (lines 14–21) specifies the invariant that should hold on the contents of the cell: namely, the cell either holds an empty value (`None`) or it contains the correct result. Then, the implementation of `memoized_computation` works by checking the cell and, if it's empty, computing the result.

```rust
pub trait InvCellPredicate<T> {
    spec fn inv(value: T) -> bool;
}

pub struct InternalInvCellPred<T, P: InvCellPredicate<T>>(T, P);

impl<T, P: InvCellPredicate<T>> InvariantPredicate<CellId, PointsTo<T>> for
    InternalInvCellPred<T, P> {
    closed spec fn inv(cell_id: CellId, points_to: PointsTo<T>) -> bool {
        points_to.id() == cell_id
          && points_to.is_init()
          && P::inv(points_to.value())
    }
}

pub struct InvCell<T, P: InvCellPredicate<T>> {
    pcell: PCell<T>,
    perm_inv: Tracked<
        LocalInvariant<CellId, PointsTo<T>, InternalInvCellPred<T, P>>
    >,
}

impl<T, P: InvCellPredicate<T>> InvCell<T, P> {
    pub closed spec fn wf(&self) -> bool {
        self.perm_inv@.constant() == self.pcell.id()
    }

    pub fn new(value: T) -> (s: Self)
        requires P::inv(value)
        ensures s.wf()
    {
        let (pcell, Tracked(points_to)) = PCell::new(value);
        let tracked local_inv = LocalInvariant::new(pcell.id(), points_to, 1337);
        InvCell { pcell: pcell, perm_inv: Tracked(local_inv) }
    }

    pub fn replace(&self, new_value: T) -> (old_value: T)
        requires
            self.wf(),
            P::inv(new_value),
        ensures
            P::inv(old_value),
    {
        let old_value;
        open_local_invariant!(self.perm_inv.borrow() => points_to => {
            old_value = self.pcell.replace(Tracked(&mut points_to), new_value);
        });
        return old_value;
    }
}
```

Figure 3.10: **Verified implementation of `InvCell`.**

```rust
pub open spec fn result_of_computation() -> u64 {
    2
}

fn expensive_computation() -> (res: u64)
    ensures
        res == result_of_computation(),
{
    1 + 1
}

pub struct MemoizationPredicate {}

impl InvCellPredicate<Option<u64>> for MemoizationPredicate {
    open spec fn inv(t_opt: Option<u64>) -> bool {
        match t_opt {
            Some(t) => t == result_of_computation(),
            None => true,
        }
    }
}

type MemoizerCell = InvCell<Option<u64>, MemoizationPredicate>;

fn memoized_computation(cell: &MemoizerCell) -> (res: u64)
    requires
        cell.wf(),
    ensures
        res == result_of_computation(),
{
    // See what value is in the cell
    let mut c = cell.replace(None);

    // If none, then perform the computation
    if c.is_none() {
        c = Some(expensive_computation());
    }

    // Put the value back in the cell for next time
    cell.replace(c);

    // Return whatever we computed or got from the cell.
    return c.unwrap();
}
```

Figure 3.11: **Verified implementation of memoizing an expensive computation.**

3.5.4 Atomics and locks

Our next example will be a mutual-exclusion lock implemented via an atomic boolean flag. To take the lock, a thread sets the atomic from `false` to `true` via an atomic `compare_exchange` instruction. Our lock will protect the access to some memory location.

The way to do this is to start with our Verus atomic primitive type, which for booleans is `PAtomicBool`. The `PAtomicBool` is a `PCell`-like type equipped with a `PointsTo`-like object, `PermissionBool`. Therefore, what we can do is put the `PermissionBool` in an `AtomicInvariant`, thus letting us access this permission across threads.

Now, we also have a memory location that our lock is trying to protect, so what do we do with that? We can take the `PointsTo` for that memory location and *also* put that in the `AtomicInvariant`. Then, when we open the `AtomicInvariant` to access the atomic permission, we can also withdraw or deposit the `PointsTo`.

In fact, this pattern—of creating an `AtomicInvariant` to store the memory permission for the atomic along with some other ghost state that the user cares about—is so common that the Verus standard library provides helper types to do it, which we collectively call the *atomic-with-ghost* library. We can use atomic-with-ghost to implement a mutual-exclusion lock (Figure 3.12).

Let's look at what's going on. We define a struct called `Lock` with two fields: an atomic boolean and a cell that stores the user's data. The atomic boolean uses the `AtomicBool` type from the atomic-with-ghost library. Much like `AtomicInvariant`, the `AtomicBool` type takes three type parameters; however, to simplify all the boilerplate involved in setting up the trait and such, the atomic-with-ghost library also provides a macro (`struct_with_invariants!`) to help set up the invariant.

The `AtomicBool` type allows us to specify a ghost type that will be "stored" in the atomic cell. On line 5, we declare that type to be an `Option<cell::PointsTo<T>>` (because there will be a `cell::PointsTo<T>` except when the lock is taken). The `struct_with_invariants!` macro helps us define an invariant that relates the physical boolean value with the ghost state. This is done in lines 12–18. Our invariant basically says: when the boolean is set to `false` (i.e., the lock is not taken), we have a `cell::PointsTo`, and when the boolean is set to `true`, we don't.

We also include the code for constructing a lock (`new`, line 23), acquiring the lock (`acquire`, line 31), and releasing the lock (`release`, line 31). To implement the last two, we use the library's `atomic_with_ghost!` macro. This macro lets us perform an atomic operation while also accessing the associated ghost state.

For example, in the `acquire` implementation, we perform a `compare_exchange` operation, attempting to atomically swap the boolean value from `false` to `true`. On line 40, we assign the ghost state (the `Option<cell::PointsTo<T>>`) to the variable `g`, which we can manipulate for the duration of the "ghost block" (lines 40–42). On line 41 we swap `g` with another variable, `points_to_opt`. This has the effect of moving the `PointsTo<T>` "out of the atomic" in the event that the atomic operation succeeds, all while preserving the relevant invariant. The proof, which the solver handles automatically, can be done by casework based on whether the value was `false` or `true`.

The implementation of `release` is a bit similar. We just store `false`, so there's no need for a loop or casework. We always set the boolean to false, and we always store the `PointsTo<T>` into the atomic.

```
struct_with_invariants!{
    struct Lock<T> {
        // The type placeholders are filled in by the
        // struct_with_invariants! macro.
        pub atomic: AtomicBool<_, Option<cell::PointsTo<T>>, _>,
        pub cell: PCell<T>,
    }

    spec fn wf(self) -> bool {
        invariant on atomic with (cell)
            is (v: bool, g: Option<cell::PointsTo<T>>)
        {
            match g {
                None => v == true,
                Some(points_to) =>
                    points_to.id() == cell.id() && points_to.is_init() && v == false,
            }
        }
    }
}

impl<T> Lock<T> {
    fn new(t: T) -> (lock: Self)
        ensures lock.wf()
    {
        let (cell, Tracked(cell_perm)) = PCell::new(t);
        let atomic = AtomicBool::new(Ghost(cell), false, Tracked(Some(cell_perm)));
        Lock { atomic, cell }
    }

    fn acquire(&self) -> (points_to: Tracked<cell::PointsTo<T>>)
        requires self.wf(),
        ensures points_to@.id() == self.cell.id(), points_to@.is_init()
    {
        loop
            invariant self.wf(),
        {
            let tracked mut points_to_opt = None;
            let res = atomic_with_ghost!(&self.atomic => compare_exchange(false, true);
                ghost g => {
                    tracked_swap(&mut points_to_opt, &mut g);
                }
            );
            if res.is_ok() {
                return Tracked(points_to_opt.tracked_unwrap());
            }
        }
    }

    fn release(&self, Tracked(points_to): Tracked<cell::PointsTo<T>>)
        requires self.wf(),
            points_to.id() == self.cell.id(), points_to.is_init()
    {
        atomic_with_ghost!(&self.atomic => store(false);
            ghost g => {
                g = Some(points_to);
            }
        );
    }
}
```

Figure 3.12: **Verified implementation of a mutual-exclusion lock.** *This example illustrates the usage of Verus's atomic-with-ghost library. The* `AtomicBool` *type, the* `struct_with_invariants!` *macro, and the* `atomic_with_ghost!` *macro are all part of this library.*

3.6 More examples and introduction to VerusSync

This section will focus on applications of VerusSync. In the first example, we will introduce the main concepts, and then introduce more advanced aspects in subsequent examples.

3.6.1 Counting to 2

The example in this section we will see how to implement and verify code like the code in Figure 3.13. (The reason I say "like" is because the actual verified code will need to need to use the Verus library types instead of the normal Rust types.) This is based on a classic example program in concurrent verification that goes all the way back to Owicki and Gries [62].

The main goal of this challenge is to verify that the assertion at the end (line 43) always passes. This should be intuitively obvious from inspection of the program, but less clear how to do it formally.

The first thing we need to do to approach a problem like this is create an abstraction of the program as a VerusSync system. By writing the VerusSync system and proving its well-formedness, Verus will create a token API that will help us solve this challenge. Our VerusSync system is shown in Figure 3.14. I will explain how to read this, then explain the token API that VerusSync generates, and then how to use the API to accomplish the verification task.

Declaring the VerusSync system At the top (lines 3–5), we declare our "state" to have 3 fields: `counter`, `inc_a`, and `inc_b`. (We will return to the "sharding" annotations in a moment.) The first corresponds to, well, the counter. The other two can each be thought of as a "right" to increment the counter. Each field starts out as `false` and can only be changed to `true` once, which can only happen when an increment is performed.

To see what I mean, take a look at the transitions. Hopefully the notation should be self-explanatory; the `new` operation shows how to initialize all the fields, while the two `transition!` operations show how to apply the increments. An `update` line updates the given field to the given value; a `require` line denotes an *enabling condition*, a condition that needs to hold in order for the transition to be applicable. The variable `pre` always refers to the pre-state of a transition.

What about the `assert` statements? We have three of them, one in `do_inc_a`, one in `do_inc_b`, and one in `finalize`, the last of which is a "property." (For now, a "property" is just an operation that can `require` and `assert` without updating anything. These will get more interesting later.) An `assert` statement is a *safety condition*, i.e., a condition needs to always hold whenever it is reached in order for the VerusSync system to be well-formed.

For example, the safety condition in `do_inc_a` says that:

- If `pre` is a reachable state,
- and if `do_inc_a` is enabled, i.e., if `!pre.inc_a` holds,
- then `pre.counter <= 2` must also hold.

(Later, this particular safety condition will be useful for showing that adding 1 does not overflow a 32-bit integer.)

Meanwhile, the `assert` in the `finalize` statement essentially shows the main thing we are trying to prove: It shows that if both increments have been performed—if both "rights" have

```rust
fn main() {
    // Initialize an atomic variable

    let atomic = AtomicU32::new(0);

    // Put it in an Arc so it can be shared by multiple threads.

    let shared_atomic = Arc::new(atomic);

    // Spawn a thread to increment the atomic once.

    let handle1 = {
        let shared_atomic = shared_atomic.clone();
        spawn(move || {
            shared_atomic.fetch_add(1, Ordering::SeqCst);
        })
    };

    // Spawn another thread to increment the atomic once.

    let handle2 = {
        let shared_atomic = shared_atomic.clone();
        spawn(move || {
            shared_atomic.fetch_add(1, Ordering::SeqCst);
        })
    };

    // Wait on both threads. Exit if an unexpected condition occurs.

    match handle1.join() {
        Result::Ok(()) => {}
        _ => { return; }
    };

    match handle2.join() {
        Result::Ok(()) => {}
        _ => { return; }
    };

    // Load the value, and assert that it should now be 2.

    let val = shared_atomic.load(Ordering::SeqCst);
    assert!(val == 2);
}
```

Figure 3.13: **Count-to-2 program as normal, unverified Rust code.**

```
CountTo2 {
    fields {
        #[sharding(variable)] pub counter: int,
        #[sharding(variable)] pub inc_a: bool,
        #[sharding(variable)] pub inc_b: bool,
    }

    init!{
        new() {
            init counter = 0;
            init inc_a = false;
            init inc_b = false;
        }
    }

    transition!{
        do_inc_a() {
            require !pre.inc_a;
            assert pre.counter <= 2;
            update counter = pre.counter + 1;
            update inc_a = true;
        }
    }

    transition!{
        do_inc_b() {
            require !pre.inc_b;
            assert pre.counter <= 2;
            update counter = pre.counter + 1;
            update inc_b = true;
        }
    }

    property!{
        finalize() {
            require pre.inc_a;
            require pre.inc_b;
            assert pre.counter == 2;
        }
    }

    #[invariant]
    pub fn main_inv(self) -> bool {
        self.counter == (if self.inc_a { 1 as int } else { 0 })
                      + (if self.inc_b { 1 as int } else { 0 })
    }
}
```

Figure 3.14: **The VerusSync system for the count-to-2 program.**

been used up—then the counter is really equal to 2.

How does Verus validate that the VerusSync system is really well-formed? In order to reason about "reachable states," Verus requires the user to supply an inductive invariant (lines 44–45). Verus checks that the invariant is actually inductive (i.e., that it holds true of any valid initial state, and that it is preserved across transitions), and it also checks that it implies all the safety conditions. In this case all these conditions happen to be straightforward, and the solver dispatches them easily. As always, more advanced cases might require manual proofs from the developer.

The token API In exchange for showing that the VerusSync system is well-formed, Verus generates a token API like the one in Figure 3.15.

First, Verus determines what *token types* to generate. This is based on the "sharding strategies" supplied in the declaration of the fields (Figure 3.14, lines 3–5). In this case, all the fields use the "variable" strategy, which means that we get one token per field. The resulting token types are `Counter`, `IncA`, and `IncB`.

Each of the 4 operations becomes a ghost function that does something with the tokens. For example, the `new` operation, which is an `init!` operation, turns into a function, `Instance::new()` that generates tokens for a fresh instantiation of the protocol. In the postcondition of this function, we can see that all the values are set to what they're initialized as in the VerusSync declaration.

The `do_inc_a` function modifies all the ghost tokens in accordance with the updates. Note that the `require` statement becomes a precondition while the `assert` statement becomes a postcondition. The `do_inc_b` and `finalize` functions are similar; observe, though, that since `finalize` does not update anything, it takes *shared references* to all the ghost tokens. This is not very important for this particular example, but it will be significant later.

There is still one crucial thing to observe. Both update operations, the function signatures of `do_inc_a` and `do_inc_b`, *only mention the tokens that are actually needed.* For example, `do_inc_a` neither observes nor updates the `inc_b` field, so it does not have to take the `IncB` token as input. This is crucial, since as we'll see in a moment, the `IncB` token will not be available when we are ready to call it.

There is also an extra type, `Instance`, which both:

- Acts as a unique identifier for an instantiation of the protocol, and
- Acts as a interface hub for all the different operations.

A verified implementation of the count-to-2 program Figure 3.16 and Figure 3.17 show a verified implementation of the count-to-2 program. Observe that we use the same atomic-with-ghost library that we introduced in §3.5.4. We create an atomic (line 4) that holds onto the `Counter` token and an invariant that makes sure the counter token's value actually matches the atomic value.

In the `main()` function, we:

- Instantiate a new instance of the protocol (line 24), obtaining an `Instance` token and tokens for the three fields.
- Initialize the atomic (line 28), giving up ownership of the `Counter` token.

```
tracked type Instance;
tracked type Counter;
tracked type IncA;
tracked type IncB;

impl Counter {
    pub spec fn instance(self) -> Instance; // Instance the token is attached to
    pub spec fn value(self) -> int;         // Value of the field
}

impl IncA {
    pub spec fn instance(self) -> Instance;
    pub spec fn value(self) -> bool;
}

impl IncB {
    pub spec fn instance(self) -> Instance;
    pub spec fn value(self) -> bool;
}

impl Instance {
    proof fn clone(tracked &self) -> (tracked res: Self)
        ensures self == res;

    proof fn new()
        -> tracked (instance: Instance, counter: Counter, inc_a: IncA, inc_b: IncB)
        ensures
            counter.instance() == instance,   counter.value() == 0,
            inc_a.instance() == instance,     inc_a.value() == false,
            inc_b.instance() == instance,     inc_b.value() == false,

    proof fn do_inc_a(tracked &self,
        tracked counter: &mut Counter, tracked inc_a: &mut IncA
    )
        requires old(counter).instance() == self, old(inc_a).instance() == self,
            old(inc_a).value() == false,
        ensures
            counter.instance() == self, inc_a.instance() == self,
            inc_a.value() == true,
            counter.value() == old(counter).value() + 1,
            old(counter).value() <= 2,

    proof fn do_inc_b(tracked &self,
        tracked counter: &mut Counter, tracked inc_b: &mut IncB
    )
        requires old(counter).instance() == self, old(inc_b).instance() == self,
            old(inc_b).value() == false,
        ensures
            counter.instance() == self, inc_b.instance() == self,
            inc_b.value() == true,
            counter.value() == old(counter).value() + 1,
            old(counter).value() <= 2,

    proof fn finalize(tracked &self,
        tracked counter: &Counter, tracked inc_a: &IncA, tracked inc_b: &IncB
    )
        requires counter.instance() == self,
            inc_a.instance() == self, inc_b.instance() == self,
            inc_a.value() == true, inc_b.value() == true,
        ensures counter.value() == 2,
}
```

Figure 3.15: **Token API generated by the `CountTo2` VerusSync system.** *Auto-generated code cleaned up for presentation.*

- Spawn the 2 threads. Pass ownership of `IncA` token to one, and the `IncB` token to the other.
  - The first thread uses the `atomic_with_ghost!` macro (line 44) to perform an atomic increment. To preserve the invariant on the atomic, we have to increment the value of the counter in the ghost block. We do this by calling `do_inc_a` (line 47).
  - The second thread does something similar.
- Join the two threads. This lets us reobtain both `IncA` and `IncB` tokens, but now both of their values are set to `true`.
- Read the value from the atomic and call `finalize` (line 102), using the fact that both tokens are `true` to deduce that the counter value is 2.

Recap The initial program seemed challenging to verify because it was impossible to have ownership over the whole program at once. To approach the problem, we found a way to factor the state into three distinct pieces—one piece for the counter, and two pieces for the implicit "roles" played by the control flow of the program's spawned threads. Even after this factorization, it remained the case that no one thread in the system could maintain ownership over all the pieces. However, we were still able to maintain key invariants about them by explicitly declaring the transitions that they were allowed to take.

```
struct_with_invariants!{
    pub struct Global {
        // An AtomicU32 that matches with the `counter` field of the ghost protocol.
        pub atomic: AtomicU32<_, Counter, _>,

        // The instance of the protocol that the `counter` is part of.
        pub instance: Tracked<Instance>,
    }

    spec fn wf(self) -> bool {
        // Specify the invariant that should hold on the AtomicU32<Counter>.
        // Specifically the ghost token (`g`) should have
        // the same value as the atomic (`v`).
        // Furthermore, the ghost token should have the appropriate `instance`.
        invariant on atomic with (instance) is (v: u32, g: Counter) {
            g.instance() == instance@
              && g.value() == v as int
        }
    }
}

fn main() {
    // Initialize protocol
    let tracked (instance, counter_token, inc_a_token, inc_b_token) = Instance::new();

    // Initialize the counter
    let tr_instance: Tracked<Instance> = Tracked(instance.clone());
    let atomic = AtomicU32::new(Ghost(tr_instance), 0, Tracked(counter_token));
    let global = Global { atomic, instance: Tracked(instance.clone()) };
    let global_arc = Arc::new(global);

    // Spawn threads

    // Thread 1
    let global_arc1 = global_arc.clone();
    let join_handle1 = spawn(
        (move || -> (new_token: Tracked<IncA>)
            ensures
                new_token@.instance() == instance && new_token@.value() == true,
            {
                // `inc_a_token` is moved into the closure
                let tracked mut token = inc_a_token;
                let globals = &*global_arc1;
                atomic_with_ghost!(&globals.atomic => fetch_add(1);
                    ghost c => {
                        // atomic increment
                        globals.instance.borrow().do_inc_a(&mut c, &mut token);
                    }
                );
                Tracked(token)
            }),
    );

    /* Continued in next figure */
```

Figure 3.16: **Verified implementation of the count-to-2 program (Part I).** *Makes use of the VerusSync ghost token API in Figure 3.15*

```
    /* Continued from previous figure */

    // Thread 2
    let global_arc2 = global_arc.clone();
    let join_handle2 = spawn(
        (move || -> (new_token: Tracked<IncB>)
            ensures
                new_token@.instance() == instance && new_token@.value() == true,
            {
                // `inc_b_token` is moved into the closure
                let tracked mut token = inc_b_token;
                let globals = &*global_arc2;
                atomic_with_ghost!(&globals.atomic => fetch_add(1);
                    ghost c => {
                        // atomic increment
                        globals.instance.borrow().do_inc_b(&mut c, &mut token);
                    }
                );
                Tracked(token)
            }),
    );

    // Join threads
    let tracked inc_a_token;
    match join_handle1.join() {
        Result::Ok(token) => {
            proof {
                inc_a_token = token.get();
            }
        }
        _ => { return; }
    }

    let tracked inc_b_token;
    match join_handle2.join() {
        Result::Ok(token) => {
            proof {
                inc_b_token = token.get();
            }
        }
        _ => { return; }
    }

    // Join threads, load the atomic again
    let global = &*global_arc;
    let x = atomic_with_ghost!(&global.atomic => load();
        ghost c => {
            instance.finalize(&c, &inc_a_token, &inc_b_token);
        }
    );

    assert(x == 2);
}
```

Figure 3.17: **Verified implementation of the count-to-2 program (Part II).**

3.6.2 Counting to n

The previous example may have seemed unsatisfactory. After all, each thread had a whole field of the VerusSync system dedicated to it. But what if we wanted to verify a program with n threads? To do this, we need to use a more advanced feature of VerusSync that allows us to create a variable number of tokens per field.

I won't go through the whole example this time, just highlight the key aspects we need to change in the VerusSync system, shown in Figure 3.18. Here are the key points:

Fixing the number of workers We have a new field, `num_workers`. For this field, the sharding strategy is something new: it is "constant." This means exactly what it sounds like—the field will not change in the system after initialization. In Figure 3.19, we see that the constant goes with the `Instance` field, so it is easy for all users of the system to agree on the value of `num_workers`.

The new `inc` field Rather than `inc_a`, `inc_b`, `inc_c`, `inc_d`, `inc_e`, ..., we have one field: `inc`. This has type `Map<int, bool>` mapping "worker IDs," that is, integers in the range [0, `num_workers`), to boolean values, each playing a role like the booleans in the previous field. Crucially, this field also uses a new sharding strategy: "map." This strategy means that, rather than getting a ghost token to represent the entire map value, we get one ghost token *per entry*. Observe how in Figure 3.19, an `Inc` token is represented by a key-value pair (`int` and `bool`).

The new transition for performing an increment Now look at the definition of `do_inc`. Because `inc` has a special sharding strategy, it can only be updated via special commands. The command,

```
remove inc -= [ i => false ]
```

says "remove the key-value pair `(i, false)` from the map." (This implicitly requires that this key-value pair be present in the map, thus establishing an enabling condition.) Meanwhile, the command,

```
add inc += [ i => true ]
```

says "add the key-value pair `(i, true)` into the map. (This implicitly asserts that the key is *not* already be present in the map, thus establishing a safety condition. In this case, the condition is trivial to prove since we just removed this key on the previous line.)

Removing and adding correspond to destroying and creating tokens, respectively. Look at the generated API method for `do_inc` (Figure 3.19, line 23). It consumes one `Inc` token (corresponding to the `remove` statement) and returns a new `Inc` token (corresponding to the `add` statement).

```
CountToN {
    fields {
        #[sharding(constant)] pub num_workers: nat,
        #[sharding(variable)] pub counter: int,
        #[sharding(map)]      pub inc: Map<int, bool>,
    }

    init!{
        initialize(num_workers: nat) {
            init num_workers = num_workers;
            init counter = 0;
            init inc = Map::new(
                |i: int| 0 <= i < num_workers,
                |i: int| false
            );
        }
    }

    transition!{
        do_inc(i: int) {
            remove inc -= [ i => false ];
            add inc += [ i => true ];
            update counter = pre.counter + 1;
            assert pre.counter < pre.num_workers;
        }
    }

    property!{
        finalize(m: Map<int, bool>) {
            // Require that 'inc' have entries [0 .. num_workers)
            // all set to true.
            have inc >= (m);
            require (forall |i: int| 0 <= i < pre.num_workers ==>
                m.dom().contains(i) && m[i] == true);

            assert(pre.counter == pre.num_workers);
        }
    }

    #[invariant]
    pub fn main_inv(&self) -> bool {
        self.inc.dom() == set_int_range(0, self.num_workers as int)
          && self.counter == num_true_in_bool_map(self.inc)
    }
}
```

Figure 3.18: **The VerusSync system for the count-to-n program.** *This VerusSync system requires a bit of proof code to show well-formedness (not shown).*

```
tracked type Instance;
tracked type Counter;
tracked type Inc;

impl Counter {
    pub spec fn instance(self) -> Instance;
    pub spec fn value(self) -> int;
}


impl Inc {
    pub spec fn instance(self) -> Instance;
    pub spec fn key(self) -> int;
    pub spec fn value(self) -> bool;
}

impl Instance {
    spec fn num_workers(self) -> nat;

    proof fn clone(tracked &self) -> (tracked res: Self)
        ensures self == res;

    proof fn do_inc(tracked &self, i: int,
        tracked counter: &mut Counter, tracked in_inc: Inc
    )
        -> (tracked out_inc: Inc)
        requires old(counter).instance() == self, in_inc.instance() == self,
            in_inc.key() == i,
            in_inc.value() == false,
        ensures
            counter.instance() == self, out_inc.instance() == self,
            out_inc.key() == i,
            out_inc.value() == true,
            counter.value() == old(counter).value() + 1;

    /* and more */
}
```

Figure 3.19: **Token API generated by the `CountToN` VerusSync system.**

3.6.3 RefCell: An application of counting permissions

In this section, we consider yet another application of a Rust cell type: Specifically, we consider `RefCell`. `RefCell` differs from `Cell` in that it allows the client to obtain *references* to the underlying type. This is challenging because we have to make sure that references do not conflict; e.g., that the client never takes out a mutable reference at the same time as a shared reference. To ensure safety, the `RefCell` internally maintains a counter that keeps track of the references that currently exist. In some regards, it is similar to a reader-writer lock; the main difference is that while a reader-writer lock will block when a conflict occurs, the `RefCell` simply fails.

Due to Verus's limitations regarding mutable references, we have to compromise on the details of the API. Specifically, we can implement and verify the following API:

```
type RefCell<S>
type Ref<'a, S>
type RefMut<'a, S>

impl<S> RefCell<S> {
    fn new(s: S) -> Self;
    fn try_borrow<'a>(&'a self) -> Option<Ref<'a, S>>;
    fn try_borrow_mut<'a>(&'a self) -> Option<RefMut<'a, S>>;
}

impl<'a, S> Ref<'a, S> {
    fn borrow<'b>(&'b self) -> &'b S;
    fn dispose(self);
}

impl<'a, S> RefMut<'a, S> {
    fn replace(&mut self, in_s: S) -> S;
    fn dispose(self);
}
```

The meaning of the `RefCell` interface The objective of a `RefCell` is to let the users take shared (immutable) references and writeable references in a way where they will never conflict—i.e., the client can never take two writeable references at once, nor can they take a writeable reference at the same time as a shared reference. Shared references, however, can be taken simultaneously.

Observe that `RefCell::try_borrow` lets the user obtain a `Ref` (if possible). The `Ref::dispose` function (usually called `drop`) destroys the `Ref`. The `RefMut` object works similarly.

Observe that `RefMut` allows mutation of the underlying object via the `replace` function. But how does `Ref` work? Let's take a close look at the type signature of the `borrow` function:

```
fn borrow<'b>(&'b self) -> &'b S
```

The `'b` is called a *lifetime variable*. In this case, it is used to tie the lifetime of the output of the reference to the output reference: Specifically, this type signature ensures that the output reference will not outlive the input reference. In particular, this means *Rust's type system makes sure that the shared reference to the underlying data cannot outlive the* `Ref`. This is, of course, crucial to the soundness of the whole interface: If the reference outlived the `Ref`, the user might then be able to obtain a `RefMut` and perform conflicting reads and writes!

Bounded lifetimes and ghost state How can we verify a function like `borrow`? What we really want a ghost type `G` supporting a a type signature like the following:

```
(&'a G) -> &'a cell::PointsTo<S>
```

If we had `&' PointsTo<S>`, we could apply something like `PCell::borrow` to get a `&'a S`.

A signature like `(&'a G) -> &'a cell::PointsTo<S>`, where a ghost shared reference to a ghost object lets you acquire a lifetime-bounded shared reference to a different ghost object, is a pattern that we will call **guarding**. So far, we haven't seen anything like this, though. How can we get it?

Verifying `RefCell` with counting permissions It turns out this is a suitable application for an old idea called *counting permissions* [5]. The idea of a counting permission system, which is traditionally expressed via separation logic, is to have two special kinds of resources: One resource is the 'counter,' and the other resource is a 'read-only permission.' The user can increase the counter to obtain a read-only permission, and they can relinquish a read-only permission to decrement the counter. When the counter is zero, one can obtain writeable permissions. Since the `RefCell` implementation *also* deals with read references and a counter thereof, it should not be surprising that they are related.

So how can we actually express counting permissions in Verus in a way that is helpful to us? It turns out we can do it with VerusSync-generated ghost state, though for this section, I'm going to explain it "in reverse": I'll start with the interface we want, and then explain how to get it with VerusSync.

Consider the interface shown in Figure 3.20–Figure 3.21. This interface is generic over a `T`. Ultimately, we're going to plug in `cell::PointsTo<S>` as `T` to get what we want.

The interface has three ghost tokens. The first one, `Instance`, should be familiar. The meat of this interface lies in the other two ghost tokens, `MainCounter` and `ReadRef`. Observe that they are all parameterized by a type parameter, `T`, the ghost object being protected (often instantiated with a `PointsTo` or similar).

The `MainCounter<T>` is represented by a value of type `Option<(nat, T)>`, which contains the counter. When the value is `None` it means that write-access is enabled, i.e., the client has ownership of the `T` value. When the `MainCounter<T>`'s representation is `Some((count, t))`, it means that the ghost object is read-only and fixed at the value `t`, while `count` represents the number of active read-references (`ReadRef` objects).

There are five crucial operations:

- **`readable_to_writeable` — Obtain ownership of `T`.** This requires the counter to be 0. The function return ghost ownership of `T` and the value of the counter is set to `None`.
- **`writeable_to_readable` — Relinquish ownership of `T`.** The caller has to give up ownership of the `T`, while the counter is set from `None` to 0.
- **`new_ref` — Obtain a new `ReadRef`.** This increments the counter and returns a ghost `ReadRef` object.
- **`delete_ref` — Destroy a `ReadRef`.** This decrements the counter and while consuming a ghost `ReadRef` object. The post condition also ensures that the counter (prior to being decremented) was ≥ 1. This follows from the fact that the `ReadRef` existed, but the client

may not have known it a priori.

- `read_ref_guards` — **Obtain a reference `&T`.** This is the aforementioned "guard" operation; observe the lifetime of the return reference is bounded by the lifetime of the input reference.

We could extend the interface with some minor features, like the ability to ensure that any two `ReadRef` objects have the same value, though we won't go into that here.

Now, informally, this ghost token API is sound because the user has to give up a `T` to get the counter to be non-None. The counter has to be set to 0 before the `T` can be "withdrawn" again. The existence of a `ReadRef` implies the counter is greater than 0. Thus, the existence of a `ReadRef` should imply the existence of a reference to the `T`. Thus, `read_ref_guards` seems like it should be sound since the lifetime of the `&T` is bounded by the input reference.

Constructing counting permissions in VerusSync We can make this train of logic more formal using VerusSync. The VerusSync code for a counting permission logic (Figure 3.22) is not particularly involved, though it does introduce several new concepts, especially the concept of *storage*.

Start by looking at the system's fields. We have three fields, `stored`, `main_counter`, and `read_ref`. The latter two, of course, correspond to the `MainCounter` and `ReadRef` types in Figure 3.20. The `main_counter` field uses the `variable` strategy, as we have seen before. The `read_ref` field uses the `multiset` strategy, which (like `map`) means that the value corresponds to a collection of tokens, one per element. (It happens that for this particular system, all `ReadRef` tokens need to agree on the underlying `T` value, which means the multiset will always be a multiple of some single element rather than an arbitrary multiset. This restriction is placed in the invariant definition.)

But what about `stored`? This field has the new strategy `storage_option`, which means that VerusSync does not generate a ghost token for this field; instead, it works with an existing ghost type, here `T`. This is why it is possible for the generated operations to interact with `T`. The `storage_option` strategy means that there will either be one thing stored or not, which is why it is represented as an `Option<T>`.

Now, let us take a look at the operations:

- `readable_to_writeable`**.** This uses the special VerusSync `withdraw` operation, which means that the client obtains ownership of some stored ghost token. The value of `stored` goes from `Some(t)` to `None` (because it is going from stored to not-stored), while in the generated API operation, the value `t` is returned.
- `writeable_to_readable`**.** This uses the `deposit` operation, which is the natural inverse of the above. The stored value changes from `None` to `Some(t)`, and the generated API operation requires the user to give up ownership of `t`.
- `new_ref`**.** The `add` operation allows us to create a new `ReadRef` token, thus adding a single element to the multiset.
- `delete_ref`**.** The `remove` operation allows us to delete a `ReadRef`, thus removing an element from the multiset.
- `read_ref_guards`**.** Note that this is a `property` rather than a `transition`. It uses two

new operation types. The `have` is a precondition that an element `t` is in the `read_ref` set. The `guard` operation is an assertion that the given value of `t` is in the `stored` field. Thus this property as a whole roughly says, "if a `ReadRef` exists, then the corresponding `t` value is stored." The `guard` operation is what allows us to generate operations with the bounded lifetimes.

As always, VerusSync generates a handful of proof obligations that are necessary for showing that the described system is well-formed. The two most important are:

- The `withdraw` operation requires that the value of `stored` in the pre-state is `Some(t)` (assuming the preconditions from the previous `require` statements).
- The `guard` operation requires that the value of `stored` is `Some(t)` (assuming the precondition from the previous `have` statement).

We need to show that these properties hold true in all reachable states of the transition system. As always, we do this by specifying an invariant, showing that it holds inductively, and that the target obligations follow from the invariant. Our invariant is given in `main_inv` (line 52). It essentially says that:

- If no element is stored, then the counter must be `None` and there are no `ReadRefs`.
- If some element is stored, then all fields agree on what that stored element is. In particular, any element in the `read_ref` multiset agrees with that element; also, the total count in the multiset matches the count from the `main_counter` field.

Fortunately, this system is simple enough that, once the invariant is specified, our solver does not need any additional proof work to check all these obligations.

```
tracked type Instance<T>;
tracked type MainCounter<T>;
tracked type ReadRef<T>;

impl<T> MainCounter<T> {
    pub spec fn instance(self) -> Instance<T>;
    pub spec fn value(self) -> Option<(nat, T)>;
}

impl<T> ReadRef<T> {
    pub spec fn instance(self) -> Instance<T>;
    pub spec fn value(self) -> T;
}

impl<T> Instance<T> {
    proof fn new() -> (tracked instance: Instance<T>, counter: MainCounter<T>)
        ensures
            counter.instance() == instance,
            counter.value() == None;

    proof fn readable_to_writeable(
        tracked &self,
        tracked counter: &mut MainCounter<T>,
    ) -> (tracked t: T)
        requires
            old(counter).instance() == self,
            match old(counter).value() {
                None => false,
                Some((count, _)) => count == 0,
            }
        ensures
            counter.instance() == self,
            counter.value() == None,
            t == old(counter).value().unwrap().1;

    proof fn writeable_to_readable(
        tracked &self,
        tracked counter: &mut MainCounter<T>,
        tracked t: T
    )
        requires
            old(counter).instance() == self,
            old(counter).value() == None,
        ensures
            counter.instance() == self,
            counter.value() == Some((0, t));

    /* Continued in next figure */
```

Figure 3.20: **A ghost state counting-permissions interface (Part I).**

```
    /* Continued from previous figure */

    proof fn new_ref(
        tracked &self,
        tracked counter: &mut MainCounter<T>,
    ) -> (tracked read_ref: ReadRef<T>)
        requires
            old(counter).instance() == self,
            old(counter).value().is_some()
        ensures
            counter.instance() == self,
            match old(counter).value() {
                None => false,
                Some((count, t)) =>
                    counter.value() == Some((count + 1, t))
                      && read_ref.value() == t
            };

    proof fn delete_ref(
        tracked &self,
        tracked counter: &mut MainCounter<T>,
        tracked read_ref: ReadRef<T>,
    )
        requires
            old(counter).instance() == self,
            old(counter).value().is_some(),
            read_ref.instance() == self,
        ensures
            counter.instance() == self,
            match old(counter).value() {
                None => false,
                Some((count, t)) =>
                    count >= 1
                      && counter.value() == Some(((count - 1) as nat, t))
            };

    proof fn read_ref_guards<'a>(
        tracked &self,
        tracked read_ref: &'a ReadRef<T>,
    ) -> (tracked borrowed_t: &'a T)
        requires
            read_ref.instance() == self,
        ensures
            borrowed_t == read_ref.value();
}
```

Figure 3.21: **A ghost state counting-permissions interface (Part II).**

```
CountingPermissions<T> {
    fields {
        #[sharding(storage_option)] pub stored: Option<T>,
        #[sharding(variable)]       pub main_counter: Option<(nat, T)>,
        #[sharding(multiset)]       pub read_ref: Multiset<T>,
    }
    init!{
        new() {
            init stored = None;
            init main_counter = None;
            init read_ref = Multiset::empty();
        }
    }
    transition!{
        readable_to_writeable() {
            require let Some((count, t)) = pre.main_counter;
            require count == 0;
            update main_counter = None;
            withdraw stored -= Some(t);
        }
    }
    transition!{
        writeable_to_readable(t: T) {
            require pre.main_counter.is_none();
            update main_counter = Some((0, t));
            deposit stored += Some(t);
        }
    }
    transition!{
        new_ref() {
            require let Some((count, t)) = pre.main_counter;
            update main_counter = Some((count + 1, t));
            add read_ref += { t };
        }
    }
    transition!{
        delete_ref(t1: T) {
            remove read_ref -= { t1 };
            require let Some((count, t2)) = pre.main_counter;
            assert count >= 1;
            assert t1 == t2;
            update main_counter = Some(((count - 1) as nat, t1));
        }
    }
    property!{
        read_ref_guards(t: T) {
            have read_ref >= { t };
            guard stored >= Some(t);
        }
    }
    #[invariant]
    pub spec fn main_inv(&self) -> bool {
        match self.stored {
            None => self.main_counter.is_none() && self.read_ref =~= Multiset::empty(),
            Some(t) => match self.main_counter {
                Some((count, t1)) => t == t1 && self.read_ref.count(t) == count
                    && (forall |t0: T| t0 != t ==> self.read_ref.count(t0) == 0),
                None => false,
            },
        }
    }
}}
```

Figure 3.22: **VerusSync for counting permissions.**

3.6.4 A reader-writer lock

A reader-writer lock is actually very similar to a `RefCell`. By combining the techniques from the previous section with our atomic-with-ghost library, we can implement a simple atomic-based reader-writer lock in a few hundred lines.

```
impl<S> RwLock<S> {
    fn new(t: T) -> RwLock<S>;

    fn read<'a>(&'a self) -> RwLockReadGuard<'a, S>;      // Read-lock acquire
    fn write<'a>(&'a self) -> RwLockWriteGuard<'a, S>;    // Write-lock acquire

    fn into_inner(self) -> S;
}

impl<'a, S> RwLockReadGuard<'a, S> {
    fn deref<'b>(&'b self) -> &'b S;
    fn drop(self);                                        // Read-lock release
}

impl<'a, S> RwLockWriteGuard<'a, S> {
    fn deref_mut<'a>(&'b mut self) -> &'b mut S;
    fn drop(self);                                        // Write-lock release
}
```

The `RwLockReadGuard` functions a lot like the `Ref` from the previous section, and likewise `RwLockWriteGuard` like a `RefMut`. Again, implementing this in Verus requires us to compromise on the mutable references, but our main focus is on the `RwLockReadGuard`. Observe that the `deref` function once again has this type signature:

```
fn deref<'b>(&'b self) -> &'b S
```

And again we solve it with a **guard** on ghost state:

```
(&'a G) -> &'a cell::PointsTo<S>
```

I'm going to consider a reader-writer lock with the following implementation. (This is kind of a simplified version of locks we consider in our case studies.) Suppose our lock state consists of two values: the *exc* boolean, which indicates if anyone has taken an exclusive write-lock, and the *rc* count, which counts how many readers have a read-lock. Also suppose they are on separate words of memory, so that they cannot be accessed by a single atomic instruction. This implementation is complicated by the fact that taking the lock is a *two-step process*.

- To take a **write lock**, a thread:
  - atomically sets *exc* from False to True.
  - waits for *rc* to reach 0.
- To take a **read lock**, a thread:
  - atomically increments *rc*
  - checks that *exc* is 0 (and if it isn't, decrement rc and start over to avoid deadlock)

In total, we have 7 transitions; 2 to take a write lock, 1 to release it; 2 to take a read lock, 1 to release it; and 1 to abandon a read-lock midway. The VerusSync system defining these transitions is shown in Figure 3.23 and Figure 3.24.

To take a write-lock, we can call `acquire_writer_start` and `acquire_writer_end`. The latter also performs a **withdraw**, giving us ownership of the stored `T`. To release the write-lock, we call `release_writer`, which performs a **deposit**.

To take a read-lock, we can call `acquire_reader_start` and `acquire_reader_end`, and to release it, we call `release_reader`. Finally, *while* we have the read-lock, we can use `read_guard` to perform a **guard** and get access to the `&T`.

```
RwLock<T> {
    fields {
        // Fields corresponding to lock's physical state
        #[sharding(variable)]       pub exc: bool,
        #[sharding(variable)]       pub rc: nat,

        // Stored (if write lock is not taken)
        #[sharding(storage_option)] pub storage: Option<T>,

        // Writer or pending writer
        #[sharding(option)]         pub pending_writer: Option<()>,
        #[sharding(option)]         pub writer: Option<()>,

        // Readers and pending readers
        #[sharding(multiset)]       pub pending_reader: Multiset<()>,
        #[sharding(multiset)]       pub reader: Multiset<T>,
    }

    init!{ ... }

    // Increment the 'rc' counter, obtain a pending_reader
    transition!{
        acquire_reader_start() {
            update rc = pre.rc + 1;
            add pending_reader += {()};
        }
    }
    // Exchange the pending_reader for a reader by checking
    // that the 'exc' bit is 0
    transition!{
        acquire_reader_end() {
            require pre.exc == false;
            remove pending_reader -= {()};
            birds_eye let x: T = pre.storage.unwrap();
            add reader += {x};
        }
    }
    // Decrement the 'rc' counter, abandon the attempt to gain
    // the 'read' lock.
    transition!{
        acquire_reader_abandon() {
            remove pending_reader -= {()};
            assert pre.rc >= 1;
            update rc = (pre.rc - 1) as nat;
        }
    }
    // Release the reader-lock. Decrement 'rc' and return the 'reader' object.
    transition!{
        release_reader(x: T) {
            remove reader -= {x};
            assert pre.rc >= 1;
            update rc = (pre.rc - 1) as nat;
        }
    }
    // Check that the 'reader' is actually a guard for the given object.
    property!{
        read_guard(x: T) {
            have reader >= {x};
            guard storage >= Some(x);
        }
    }
```

Figure 3.23: **VerusSync for a simple reader-write lock (Part I).**

```
    // Atomically set 'exc' bit from 'false' to 'true'
    // Obtain a pending_writer
    transition!{
        acquire_writer_start() {
            require pre.exc == false;
            update exc = true;
            add pending_writer += Some(());
        }
    }

    // Finish obtaining the write lock by checking that 'rc' is 0.
    // Exchange the pending_writer for a writer and withdraw the
    // stored object.
    transition!{
        acquire_writer_end() {
            require pre.rc == 0;
            remove pending_writer -= Some(());
            add writer += Some(());
            birds_eye let x = pre.storage.unwrap();
            withdraw storage -= Some(x);
        }
    }

    // Release the write-lock. Update the 'exc' bit back to 'false'.
    // Return the 'writer' and also deposit an object back into storage.
    transition!{
        release_writer(x: T) {
            remove writer -= Some(());
            update exc = false;
            deposit storage += Some(x);
        }
    }

    #[invariant]
    pub spec fn exc_bit_matches(&self) -> bool {
        (if self.exc { 1 } else { 0 as int }) ==
            (if self.pending_writer.is_some() { 1 } else { 0 as int }) as int
            + (if self.writer.is_some() { 1 } else { 0 as int }) as int
    }

    #[invariant]
    pub spec fn count_matches(&self) -> bool {
        self.rc == self.pending_reader.count(())
            + self.reader.count(self.storage.unwrap())
    }

    #[invariant]
    pub spec fn reader_agrees_storage(&self) -> bool {
        forall |t: T| self.reader.count(t) > 0 ==>
            self.storage == Option::Some(t)
    }

    #[invariant]
    pub spec fn writer_agrees_storage(&self) -> bool {
        self.writer.is_some() <==> self.storage.is_none()
    }
}
```

Figure 3.24: **VerusSync for a simple reader-write lock (Part I).**

3.7 VerusSync recap: key points

VerusSync introduced us to a couple of key points that should be emphasized, as they will help motivate our formalization efforts going forward.

3.7.1 "Prove global properties, export local features"

We saw this principle in every VerusSync system we looked at so far. In CountTo2 (§3.6.1), we had 3 fields, but some of the key operations only involved 2 of the tokens. They were *local* in the sense that they could be performed as long as we had the relevant state. However, the key invariants of the system, needed to show the well-formedness conditions, were stated as invariant predicates over the global state.

In our counting permissions and RwLock examples, we saw this principle manifest through the **guard** statements. Proving the **guard** commands well-formed required global invariant predicates, but the exported guard properties operate on single tokens.

3.7.2 The deposit/withdraw/guard pattern

We saw a certain pattern come up whenever we needed to coordinate shared read-access and exclusive write-access to some ghost object `T`.

Specifically, the client could **deposit** the `T` to give up exclusive ownership (though we were somewhat vague on what, exactly, we were 'depositing into'). We also saw that it could **withdraw** the `T` to regain exclusive ownership. We also saw that, in the interim, the client could obtain shared references to the `T`, a process we called **guarding**. We also asserted that VerusSync is somehow able to check that this protocol is *well-formed* in the sense that the guarding will not outlast the point of withdrawal.

This pattern is one we will call the **deposit/withdraw/guard** pattern, and it will come up frequently in the remainder of this document.

Operation	Signature of ghost function
Deposit	`fn deposit(tracked T, ...)`
Withdraw	`fn withdraw(...) -> tracked T`
Guard	`fn guard<'a>(tracked &'a S) -> tracked &'a T`

Both examples we looked at, `RefCell` and `RwLock`, had obvious guard-like signatures in their interfaces, and these motivated the deposit/withdraw/guard pattern. However, since Verus expresses this pattern through *ghost* objects, we do not necessarily need a *physical* guard object to apply the pattern.

As a comparison, consider the doubly-linked list example, where we saw that we could use ghost state to create our own ownership structure decoupled from the physical pointer structure. In the same way, we can use ghost objects to engineer new guarding structures even in situations where the code doesn't have such blatant guard objects. We will appreciate this point more when we return to our case studies.

3.8 Chapter Recap

This concludes the "example-driven" portion of the explanation, which hopefully gives an idea of how the different pieces can fit together in interesting ways, and of what kinds of things ghost state is capable.

Of course, the reader must be left with several questions. Just what exactly *is* VerusSync doing? What does it mean for a VerusSync system to be well-formed? What is the formal basis for the deposit/withdraw/guard pattern, and how can we ensure that this disparate set of primitive types is really sound as a whole? We will answer these questions in the coming chapters.

Chapter 4

Ghost State as Monoids

We will now begin our journey to formally characterize and justify the primitive features of Verus, including pointers, cells, the invariant types, and user-defined ghost state with VerusSync. We will especially emphasize user-defined ghost state and its interactions with shared references in Rust's type system, as this is where most of the novelty lies.

Because VerusSync is somewhat complex, we need to factor this process into several steps. First, we will start by defining an alternative to VerusSync called the **Verus Monoidal Ghost Interface**. As an interface for constructing ghost state, it is more technically challenging to use than VerusSync, but it is more "canonical" and easier to formalize.

Our roadmap is as follows:

- In this chapter, we will define the Verus Monoidal Ghost Interface, inspired by monoid-based ghost interfaces in the Iris concurrent separation logic. In doing so, we will also cover technical Iris material needed for subsequent chapters.
- In Chapter 5 we will formally describe VerusSync and justify it by a "lowering" into the Verus Monoidal Ghost Interface.
- In Chapter 6 we will define a formal type system and specification system which covers shared borrows, lifetimes, and Verus primitives including pointers, cells, invariants, and the Verus Monoidal Ghost Interface.

4.1 The Verus Monoidal Ghost Interface: High-level picture

In this chapter, we are going to define the Verus Monoidal Ghost Interface, inspired by ghost state laws from the Iris separation logic. To build up Verus Monoidal Ghost Interface incrementally, I will present two versions of the interface: the *resource algebra interface* and the *storage protocol interface*.

I will start by reviewing resource algebras in Iris and use them to construct the resource algebra interface. However, this interface will leave several important elements unaddressed. This will motivate the need for a more elaborate ghost state theory. I will present such a theory via my Iris library *Leaf*. Finally, I will use this new theory to define the storage protocol interface.

4.2 Background: The Iris Separation Logic

Iris [35] is a framework for concurrent separation logic. As a result, it admits Hoare-style specifications and Hoare-style reasoning principles, but most importantly, Iris is a logic of *resources*, and in this chapter, it is mostly the logic of resources that concerns us.

The Iris resource logic, or "base logic" as it is usually called, is composed of the connectives of bunched implications [63]—the separating conjunction ($*$), the magic wand ($-\!\!*$), the non-separating conjunction ($\wedge$), disjunction ($\vee$)—and a small number of modalities. Beyond the base logic, there are also a handful of derived modalities.

Updates For us, the most important modality is the *update modality*, written $\Rightarrow\!\!\Rightarrow Q$. There are a variety of notations for describing different kinds of updates. We will need a lot of them for a variety of different reasons, but it will be easiest to just cover them all in one place.

- $P \Rrightarrow Q$ is called a *view shift*, the knowledge that we can always exchange a resource P for Q.
- $P \Rrightarrow\!\!\!\!* \; Q$ is the *view shift wand*, it represents the right to exchange P for Q. Unlike $P \Rrightarrow Q$, though, can only be used once (similar to the $-\!\!*$).

We can also annotate the updates with masks. These are called *fancy updates*.

- $P \;\; {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} \; Q$ is the *mask-changing view shift*, which not only exchanges P for Q but updates the "mask" from $\mathcal{E}_1$ to $\mathcal{E}_2$.
- $P \; {}^{\mathcal{E}_1}\!\!\Rrightarrow\!\!\!\!*^{\mathcal{E}_2} \; Q$ is a mask-changing update that can only be applied once.

A mask is a set of names that denotes the set of "openable invariants." This is best explained through the invariant-opening rule. (Ignore the $\triangleright$ for a moment.)

$$\textsc{Inv-Open}\quad \frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash \mathsf{True} \;\; {}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \; (\triangleright P) * ((\triangleright P) \; {}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\!\!*^{\mathcal{E}} \; \mathsf{True})}$$

The way to read this is as follows: If $\boxed{P}^{\mathcal{N}}$ holds (that is, if P is an invariant in the namespace $\mathcal{N}$), and if $\mathcal{N}$ is contained in the current mask, then it possible to "open" the invariant. This is done by a view shift ${}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}}$ that gives us ownership of P. By performing this view shift, the mask no longer contains $\mathcal{N}$, so the same invariant cannot be double-opened. We also get a view shift wand, ${}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\!\!*^{\mathcal{E}}$ which requires us to relinquish ownership of P in order to close the invariant.

(The reader may notice this is actually just like the system we described for Verus (§3.4.5), and in fact, it is the direct inspiration for Verus's system. That said, this is not really the reason we bring it up here; we are going to be using invariants and masks for somewhat tangential reasons.)

Later Modalities The other significant modality in the Iris separation logic is the *later modality*, written $\triangleright P$. The later modality is needed for technical soundness reasons, though its semantic meaning is somewhat hard to describe. What the reader needs to know is:

- There are some situations that require us to add a later ($\triangleright$) in front of a proposition (like the Inv-Open rule above).
- We can strip off these $\triangleright$ modalities in a limited way; Iris usually sets up its program logic so that some bounded number of $\triangleright$ modalities can be removed at each program step.

The later modality is often regarded as "a necessary evil." Customarily, we will try to ignore it as much as we get away with.

4.3 Resource Algebras

The Iris separation logic has developed the concept of a *resource algebra (RA).* a mechanism that allows the user to construct new "resources" and derive updates and deductions. The key idea is that if we, as users of the Iris logic, write down an RA and show that it satisfies all the RA laws, then we get access to a resource we can manipulate within the logic.

In this section, I will give a rough overview of how this works. Specifically, I will present a special case called a "unital RA." Formally, a unital resource algebra is a *monoid*, i.e., a set M with a composition operator $\cdot : M \times M \to M$ which is associative and commutative, and with a unit element ϵ, together with a validity predicate $\mathcal{V} : M \to \mathit{Prop}$. The validity predicate must be *closed under inclusion*, that is, if we let, $a \preccurlyeq b \triangleq (\exists c.\ a \cdot c = b)$, then $\mathcal{V}$ must satisfy: $\forall a, b.\ a \preccurlyeq b \land \mathcal{V}(b) \Rrightarrow \mathcal{V}(a)$. Furthermore, the unit is always valid, i.e., $\mathcal{V}(\epsilon)$ holds.

With the resource algebra in hand, we next define a derived relation called a *frame-preserving update*.

$$a \rightsquigarrow b \triangleq \forall c.\ \mathcal{V}(a \cdot c) \Rrightarrow \mathcal{V}(b \cdot c).$$

Essentially, a can transition to b if for any valid way of "completing" state, the state would remain valid after the transition.

For any such M, Iris provides a ghost resource denoted $\boxed{a}^\gamma$ for any $a : M$, together with the proof rules in Figure 4.2. The $\gamma : \mathit{Name}$ is a *ghost name* (sometimes called *ghost location*) from an arbitrary, infinite set of available names. These rules show, for instance, that the compositional structure $(\cdot)$ of the monoid determines the compositional structure within the logic, i.e., $\boxed{a \cdot b}^\gamma$ is equivalent to $\boxed{a}^\gamma * \boxed{b}^\gamma$. Likewise, an update $a \rightsquigarrow b$ means that we can exchange $\boxed{a}^\gamma$ for $\boxed{b}^\gamma$ as resources within the logic: $\boxed{a}^\gamma \Rrightarrow \boxed{b}^\gamma$ as given by RA-Update.

In other words, by defining a resource algebra, we not only get a bunch of compositional resources, we get a bunch of updates we can perform that are guaranteed to maintain the overall validity of the resources.

Example 1 *An archetypal Resource Algebra is the* exclusive monoid, *$\textsc{Excl}(X)$, for a given set X. The elements of $\textsc{Excl}(X)$ are made out of the following symbols:*

$$\epsilon \mid \mathit{ex}(x) \mid \lightning \qquad \text{with } \forall x, y.\ \mathit{ex}(x) \cdot \mathit{ex}(y) = \lightning \text{ and } \forall a, a \cdot \epsilon = a \text{ and } a \cdot \lightning = \lightning$$

Here, ϵ is the unit element, representing ownership of nothing, the value $\mathit{ex}(x)$ represents exclusive ownership of a state x, and $\lightning$ represents the impossible "conflict" state of multiple ownership claims. The elements ϵ and $\mathit{ex}(x)$ are all considered "valid," while $\lightning$ is "invalid," i.e., $\mathcal{V}(\lightning) = \mathit{False}$. One can show that for any $x, y : X$, $\mathit{ex}(x) \rightsquigarrow \mathit{ex}(y)$, which implies the view shift, $\boxed{\mathit{ex}(x)}^\gamma \Rrightarrow \boxed{\mathit{ex}(y)}^\gamma$ by RA-Update. That is, given ownership of the state, one can freely update it.

A **unital resource algebra** consists of a set M, a composition operator $\cdot : M \times M \to M$, and a validity predicate $\mathcal{V} : M \to \mathit{Prop}$ satisfying:

$$\begin{aligned}
&\forall a.\ a \cdot \epsilon = a \\
&\forall a, b.\ a \cdot b = b \cdot a \\
&\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \\
&\mathcal{V}(\epsilon) \\
&\forall a, b.\ a \preceq b \land \mathcal{V}(b) \Rightarrow \mathcal{V}(a)
\end{aligned}$$

where:

$$\begin{aligned}
a \preccurlyeq b &\triangleq (\exists c.\ a \cdot c = b) \\
a \rightsquigarrow b &\triangleq \forall c.\ \mathcal{V}(a \cdot c) \Rightarrow \mathcal{V}(b \cdot c) \\
a \rightsquigarrow B &\triangleq \forall c.\ \mathcal{V}(a \cdot c) \Rightarrow \exists b.\ b \in B \land \mathcal{V}(b \cdot c)
\end{aligned}$$

Figure 4.1: **Definition of a unital resource algebra.** *This is slightly simplified: The standard definition [35] also includes something called a* core*, which we elide as it will not come up for our purposes.*

Rules for RA-based ghost state
Instantiated for a given unital RA $(M, \cdot, \mathcal{V})$
Propositions: $\boxed{a}^{\gamma}$ (where $a : M$, γ : *Name*)

RA-Unit
$\mathsf{True} \vdash \boxed{\epsilon}^{\gamma}$

RA-Valid
$\boxed{a}^{\gamma} \vdash \mathcal{V}(a)$

RA-Sep
$\boxed{a \cdot b}^{\gamma} \dashv\vdash \boxed{a}^{\gamma} * \boxed{b}^{\gamma}$

RA-Alloc
$$\frac{\mathcal{V}(a)}{\mathsf{True} \Rrightarrow \exists \gamma.\ \boxed{a}^{\gamma}}$$

RA-Update
$$\frac{a \rightsquigarrow b}{\boxed{a}^{\gamma} \Rrightarrow \boxed{b}^{\gamma}}$$

RA-Update-Nondeterministic
$$\frac{a \rightsquigarrow B}{\boxed{a}^{\gamma} \Rrightarrow \exists b.\ (b \in B) * \boxed{b}^{\gamma}}$$

RA-And
$\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \vdash \exists c.\ \boxed{c}^{\gamma} * (a \preccurlyeq c) * (b \preccurlyeq c)$

Figure 4.2: **Iris rules for RA-based ghost state.**

Nondeterministic updates The frame-preserving update $a \rightsquigarrow b$ uses a fixed b. It is also possible to use a *set* of possible result values, i.e. for a set $B \subseteq M$, we can define,

$$a \rightsquigarrow B \triangleq \forall c.\ \mathcal{V}(a \cdot c) \Rightarrow \exists b.\ b \in B \wedge \mathcal{V}(b \cdot c)$$

Effectively, this allows us to transition to a value that depends on c. The nondeterministic update is used in the rule RA-Update-Nondeterministic.

Using the overlapping conjunction One slightly unusual rule here is the RA-And rule which lets us combine ghost state via the non-separating conjunction $\wedge$.[1] As an example, consider $\boxed{\mathsf{ex}(x)}^\gamma \wedge \boxed{\mathsf{ex}(y)}^\gamma$. If $x \neq y$, then $\boxed{\mathsf{ex}(x)}^\gamma \wedge \boxed{\mathsf{ex}(y)}^\gamma \vdash \boxed{\lightning}^\gamma$ by RA-And (since $\lightning$ is the only possible value of c which is $\succcurlyeq$ both x and y. Then (since $\boxed{\lightning}^\gamma \vdash \mathsf{False}$) we can conclude that $\boxed{\mathsf{ex}(x)}^\gamma \wedge \boxed{\mathsf{ex}(y)}^\gamma \vdash (x = y)$.

4.4 Resource Algebras in Verus

Our first step is to "translate" resource algebra-based ghost state into a Verus interface of ghost objects.

We start by defining RA as a trait (Figure 4.3). The Verus developer can provide a specific unital RA by implementing the trait, which requires them to provide three spec functions: `valid` (corresponding to $\mathcal{V}$), `op` (corresponding to $\cdot$), and `unit` (corresponding to ϵ); they then provide proofs that the RA meets all the RA conditions like commutativity and so on. (These proofs are usually trivial for simple RAs, with the SMT solver able to solve them automatically.)

In exchange for developing some type `P` which implements `RA`, the developer can use the ghost object `Resource<P>`, the Verus analogue of $\boxed{x}^\gamma$. The `Resource<P>` has two pieces of data: the ghost location (corresponding to γ) and the ghost value (corresponding to x). Figure 4.4 shows the operations you can perform on these objects, which correspond to the classic rules in Figure 4.2.

For example, let us take a look at `Resource::alloc` in detail. `Resource::alloc` corresponds to RA-Alloc. RA-Alloc lets the developer pick some a satisfying $\mathcal{V}(a)$ and obtain the ghost state $\boxed{a}^\gamma$. The user does not get to specify γ, though, which is chosen arbitrarily. Likewise, the function `Resource::alloc` takes one input, the value `a`. It returns the ghost object with `value` set to `a` and some arbitrary location.

Likewise, we have rules corresponding to RA-Sep, RA-Unit, RA-Update, and RA-Update-Nondeterministic.

Now, since we are working in Rust, we can also take *shared references* to these ghost objects. Figure 4.5 shows what we can do with shared references to ghost state. Individually:

- In `join_shared`, we cannot compose the way we normally would compose ghost state with $*$ and RA-Sep, because the ghost objects might not actually be separated. Instead, we treat them as if they were composed by $\wedge$ and use the condition from RA-And.

[1] I presented a similar rule in the Leaf paper [26]; the version given here is slightly more general. Proofs for both can be found in Leaf's Coq formalization. The proof is fairly straightforward, and it is not dependent on the main results of Leaf.

```
pub trait RA {
    // Definition of a Resource Algebra

    spec fn valid(self) -> bool;           // 𝒱
    spec fn op(self, other: Self) -> Self; // ·
    spec fn unit() -> Self;                // ϵ

    // Well-formedness conditions for a resource algebra (Figure 4.1)

    proof fn closed_under_incl(a: Self, b: Self)
        requires
            Self::op(a, b).valid(),
        ensures
            a.valid();

    proof fn commutative(a: Self, b: Self)
        ensures
            Self::op(a, b) == Self::op(b, a);

    proof fn associative(a: Self, b: Self, c: Self)
        ensures
            Self::op(a, Self::op(b, c)) == Self::op(Self::op(a, b), c);

    proof fn op_unit(a: Self)
        ensures
            Self::op(a, Self::unit()) == a;

    proof fn unit_valid()
        ensures
            Self::valid(Self::unit());
}

// Definition of ≼
pub open spec fn incl<P: RA>(a: P, b: P) -> bool {
    exists|c| P::op(a, c) == b
}

// Definition of a ⇝ b
pub open spec fn frame_preserving_update<P: RA>(a: P, b: P) -> bool {
    forall|c| P::op(a, c).valid() ==> P::op(b, c).valid()
}

// Definition of a ⇝ B
pub open spec fn frame_preserving_update_nondeterministic<P: RA>(
    a: P, bs: Set<P>
) -> bool {
    forall|c|
        P::op(a, c).valid() ==> exists|b| bs.contains(b) && P::op(b, c).valid()
}
```

Figure 4.3: **Verus's ghost state encoding of a Resource Algebra (Part I).**

```
// Ghost state representing ⌜x⌝^γ
pub tracked type Resource<P>

impl<P: RA> Resource<P> {
    // Spec encoding of a Resource
    pub open spec fn value(self) -> P;  // x
    pub open spec fn loc(self) -> Loc;  // γ

    // Operations on resources

    // RA-Alloc
    pub proof fn alloc(a_value: P) -> (tracked a: Self)
        requires
            a_value.valid(),
        ensures
            a.value() == a_value;

    // RA-Sep (going backward)
    pub proof fn join(tracked a: Self, tracked b: Self) -> (tracked out: Self)
        requires
            a.loc() == b.loc(),
        ensures
            out.loc() == a.loc(),
            out.value() == P::op(a.value(), b.value());

    // RA-Sep (going forward)
    pub proof fn split(tracked self, a_value: P, b_value: P)
        -> (tracked out: (Self, Self))
        requires
            self.value() == P::op(a_value, b_value),
        ensures
            out.0.loc() == self.loc(),
            out.1.loc() == self.loc(),
            out.0.value() == a_value,
            out.1.value() == b_value;

    // RA-Unit
    pub proof fn create_unit(loc: Loc) -> (tracked out: Self)
        ensures
            out.value() == P::unit(),
            out.loc() == loc;

    // RA-Valid
    pub proof fn validate(tracked a: &Self)
        ensures a.value().valid();

    // RA-Update
    pub proof fn update(tracked a: Self, b_value: P) -> (tracked b: Self)
        requires frame_preserving_update(a.value(), b_value),
        ensures
            b.loc() == a.loc(),
            b.value() == b_value;

    // RA-Update-Nondeterministic
    pub proof fn update_nondeterministic(tracked a: Self, b_values: Set<P>)
        -> (tracked b: Self)
        requires frame_preserving_update_nondeterministic(a.value(), b_values),
        ensures
            b.loc() == a.loc(),
            b_values.contains(b.value());
}
```

Figure 4.4: **Verus's ghost state encoding of a Resource Algebra (Part II).**

```
// {q · t | q ∈ set}
pub open spec fn set_op<P: RA>(set: Set<P>, t: P) -> Set<P> {
    Set::new(|v| exists|q| set.contains(q) && v == P::op(q, t))
}

impl<P: RA> Resource<P> {
    // Operations with shared references

    // RA-And
    pub proof fn join_shared<'l>(
        tracked a: &'l Self,
        tracked b: &'l Self,
    ) -> (tracked c: &'l Self)
        requires
            a.loc() == b.loc(),
        ensures
            c.loc() == a.loc(),
            incl(a.value(), c.value()),
            incl(b.value(), c.value()),

    pub proof fn weaken<'l>(tracked &'l self, target: P) -> (tracked out: &'l Self)
        requires incl(target, self.value()),
        ensures out.loc() == self.loc(), out.value() == target;

    // RA-Valid, but where part is shared
    pub proof fn validate_with_shared(tracked &mut a: Self, tracked b: &Self)
        requires
            old(self).loc() == other.loc(),
        ensures
            *self == *old(self),
            P::op(self.value(), other.value()).valid();

    // RA-Update, but where part is shared
    pub proof fn update_with_shared(
        tracked a: Self,
        tracked x: &Self,
        b_value: P,
    ) -> (tracked b: Self)
        requires
            a.loc() == x.loc(),
            // (a · x) ⇝ (b · x)
            frame_preserving_update(
                P::op(a.value(), x.value()),
                P::op(b_value, x.value())),
        ensures
            b.loc() == a.loc(),
            b.value() == b_value;

    // RA-Update-Nondeterministic, but where part is shared
    pub proof fn update_nondeterministic_with_shared(
        tracked a: Self,
        tracked x: &Self,
        b_values: Set<P>,
    ) -> (tracked b: Self)
        requires
            a.loc() == x.loc(),
            frame_preserving_update_nondeterministic(
                P::op(a.value(), x.value()),
                set_op(b_values, x.value())),
        ensures
            b.loc() == a.loc(),
            b_values.contains(b.value());
}
```

Figure 4.5: **Verus's ghost state encoding of a Resource Algebra (Part III).**

- In `weaken`, we can turn the value into any value $\preccurlyeq$ to it.
- In `validate_with_shared`, we validate the composition of two ghost state objects, when one is owned exclusively and the other is owned shared. (Note that we cannot do this if *both* are shared, since they might overlap!)
- In `update_with_shared`, we perform an *update* even if part of it shared. The idea is that if we have $(a \cdot x) \rightsquigarrow (b \cdot x)$ then we should be able to perform the transition even if x is shared and read-only. We just leave the x part un-touched and change a to b.
- Finally, `update_nondeterministic_with_shared` is just the nondeterministic version of `update_with_shared`.

Now, while these rules should make some intuitive sense, it is not immediately clear exactly what "shared references to ghost state" are supposed to correspond to when you look back at the CSL inspiration of $\boxed{x}^{\gamma}$. How do we justify the claim that combining two shared references is the same thing as $\wedge$, or that a frame-preserving updates can be performed when part of the state is behind a shared reference? Finally, how do we create borrows with bounded lifetimes, e.g., `(&'a T) -> &'a S`, which as we saw in Chapter 3 is useful as part of the deposit/withdraw/guard pattern? Right now, there is no clear way to do that at all.

Answering these questions is the primary motivation for the developments in the next section.

4.5 Leaf: A generalization of read-write permission logics

Leaf [26] is a separation logic libary I developed on top of Iris, designed as an abstraction to handle "temporarily shared, read-only resources." There were roughly two semi-orthogonal motivations for the development of Leaf:

- To provide a formal basis for the ghost axioms used in the IronSync/Verus methodology.
- To generalize the concept of "fractional permissions" [6] and "counting permissions" [5] in separation logic.

Of course, I will focus primarily on the first motivation in this thesis.

Fundamentally, the question Leaf addresses is: *How can we represent, apply, and manipulate read-only shared state in separation logic?* Our answer to this question was motivated by a few key observations:

- Traditionally, separation logic handles temporarily shared, read-only state via either *fractional permissions* or *counting permissions*. (In fact, even counting permissions seem to be somewhat rare; usually fractional permissions are used.) However, while these are simple and elegant, they do not capture the complexity and breadth of real read-sharing strategies (see Challenge SpC-2 and Challenge NR-1). This observation prompts us to ask if these can be generalized.
- Fractional and counting permissions each work, fundamentally, by having some sub-structural permission that represents a read-only version of some other permission. For example, in fractional permissions, the "fractional points-to" $\ell \overset{\text{frac}}{\hookrightarrow}_q v$ represents a read-only version of the full points-to $\ell \hookrightarrow v$. Counting permissions works similarly, except that

it is a *different* permission, the "read-only points-to," $\ell \overset{\text{ro}}{\hookrightarrow} v$ that represents a read-only $\ell \hookrightarrow v$.

In light of the second point, Leaf posits that the key property behind a read-only permission system is the ability for one kind of permission to stand in as the read-only version of an otherwise exclusive permission. The key challenge becomes, then, to come up with a *uniform* way to represent this idea that can capture both the $\ell \overset{\text{frac}}{\hookrightarrow}_q v$ and $\ell \overset{\text{ro}}{\hookrightarrow} v$.

To do this, Leaf introduces a novel operator, $\twoheadrightarrow$, pronounced "guards," which represents the relationship between a substructural proposition and the proposition that it represents the sharing of.

Leaf is formalized in Coq. In this thesis, I will not be detailing the construction of the $\twoheadrightarrow$ operator or the proofs of Leaf's deduction rules, though there is an outline of the construction in the Leaf paper [26], and the proofs are of course available in the Coq formalization. This section will also introduce some new rules not present in the Leaf paper, some of which are used to simplify the presentation, and some of which are needed for Chapter 6. These new rules, too, are available in the Coq formalization.

4.5.1 An introduction to the guards operator

The primary question we that we unravel in Leaf is how to talk generally about "a shared P" for any proposition P. Here, the proposition P might be something simple, like the memory permission $\ell \hookrightarrow v$, some other simple resource like $\boxed{x}^{\gamma}$, or even a more complex invariant.

We do this via a relationship $G \twoheadrightarrow P$, pronounced G *guards* P. $G \twoheadrightarrow P$ is itself a proposition; informally, it means that G can be used as a "shared P." Hence, if some program proof needs to operate over a "shared P," it can instead take G as an exclusively owned precondition, and use the relationship $G \twoheadrightarrow P$ when it needs to use P. Later, G might be consumed (disallowing further shared access to P), and eventually the exclusive ownership of P might be reclaimed.

Leaf is a library about abstract resources, but of course the motivation for it is to be used in program logics. Let's take a look at a common situation. Suppose, for example, that we have a program and some resource P, and that we want the program's specification to require the resource P in a shared, read-only fashion. Suppose further that we want to write the specification *generically* over the means with which P is shared. To do so, we can write the proof to take ownership of some arbitrary Iris proposition G : *iProp* where $G \twoheadrightarrow P$. To codify this pattern, we use a shorthand, $[X]\ \{P\}\ e\ \{Q\}$, to mean $\forall G : \textit{iProp}.\ \{P * G * (G \twoheadrightarrow X)\}\ e\ \{Q * G\}$. This can be read as "if command e executed, with P owned at the beginning, and with X shared, then Q is owned at the end."

For example, a program logic might allow writing to a memory location given exclusive ownership of $\ell \hookrightarrow v$, but allow reading from it given *shared* ownership of $\ell \hookrightarrow v$. Leaf specifies this as:

Heap-Write	Heap-Read-Shared
$\{\ell \hookrightarrow v\}\ \ell \leftarrow v'\ \{\ell \hookrightarrow v'\}$	$(G \twoheadrightarrow_{\mathcal{E}} (\ell \hookrightarrow v)) \vdash \{G\}\ !\ell\ \{r.\ (v = r) * G\}$

Here, $\ell \leftarrow v'$ is the command to write to the reference ℓ, while $!\ell$ reads it. A bound variable in a postcondition, e.g. r here, represents the command's return value, so Heap-Read-Shared says, if we have a shared $\ell \hookrightarrow v$ and read from ℓ, then we obtain a value equal to v.

RwLock Specification

Propositions: $\mathsf{IsRwLock}(rw, \gamma, F)$ $\quad \mathsf{Exc}(\gamma)$ $\quad \mathsf{Sh}(\gamma, x)$

(where $rw : \mathit{Value}$, $\gamma : \mathit{Name}$, $X : \mathit{Set}$, $x : X$, $F : X \to \mathit{iProp}$)

$\forall F, x.$		$\{F(x)\}$ rwlock_new()	$\{rw.\ \exists \gamma, \mathsf{IsRwLock}(rw, \gamma, F)\}$
$\forall rw, \gamma, F.$		$\{\mathsf{IsRwLock}(rw, \gamma, F)\}$ rwlock_free(rw)	$\{\}$
$\forall rw, \gamma, F.$	$[\mathsf{IsRwLock}(rw, \gamma, F)]$	$\{\}$ lock_exc(rw)	$\{\mathsf{Exc}(\gamma) * \exists x.\ F(x)\}$
$\forall rw, \gamma, F, x.$	$[\mathsf{IsRwLock}(rw, \gamma, F)]$	$\{\mathsf{Exc}(\gamma) * F(x)\}$ unlock_exc(rw)	$\{\}$
$\forall rw, \gamma, F.$	$[\mathsf{IsRwLock}(rw, \gamma, F)]$	$\{\}$ lock_shared(rw)	$\{\exists x.\ \mathsf{Sh}(\gamma, x)\}$
$\forall rw, \gamma, F, x.$	$[\mathsf{IsRwLock}(rw, \gamma, F)]$	$\{\mathsf{Sh}(\gamma, x)\}$ unlock_shared(rw)	$\{\}$

$\forall rw, \gamma, F, x.\ \mathsf{IsRwLock}(rw, \gamma, F) \vdash (\mathsf{Sh}(\gamma, x) \twoheadrightarrow F(x))$

Figure 4.6: **Leaf-style specification for a reader-writer lock.** *Explored in depth in the Leaf paper [26].*

Using the bracket notation, Heap-Read-Shared could be written:

Heap-Read-Shared
$[\ \ell \hookrightarrow v\ ]\{\}\ !\ell\ \{r.\ (v = r)\}$

4.5.2 Example: A spec for a reader-writer lock

The reader-writer lock spec in Figure 4.6 illustrates several facets of our guarding system. The API of this lock has six functions: rwlock_new and rwlock_free are the constructor and destructor, respectively; lock_exc and unlock_exc are intended to allow exclusive, write access to some underlying resource; lock_shared and unlock_shared are intended to allow shared, read-only access. Exactly what this "resource" is may be determined by the client.

Holding the spec together is the proposition $\mathsf{IsRwLock}(rw, \gamma, F)$, which roughly says that the value rw is a reader-writer lock with a unique identifier γ. F is used to specify the resource being protected—we will return to this in a moment. Note that when a new reader-writer lock is constructed (via rwlock_new) the client obtains exclusive ownership over $\mathsf{IsRwLock}(rw, \gamma, F)$; on the other hand, the operations that are meant to run concurrently all take $\mathsf{IsRwLock}(rw, \gamma, F)$ as *shared*. The destructor, rwlock_free, again requires non-shared ownership, as naturally it should not be able to run concurrently with other operations.

Now, the client needs to specify what sort of resource they want to protect. For example, the client might want to protect access to some location in memory, say ℓ, so they would use the lock to protect resources of the form $\ell \hookrightarrow v$. To allow the client to choose the kind of resource they want to protect, our specification lets the client, upon construction of the lock, provide a *proposition family* $F : X \to \mathit{iProp}$ parameterized over some set X. In the above example, we might have $F = \lambda x : \mathit{Value}.\ \ell \hookrightarrow x$ for some fixed ℓ determined at the time of the rwlock_new() call.

In the specification, observe that we then use $F(x)$, for some x, to represent the resource when it is obtained from the lock by lock_exc. Upon calling unlock_exc, the client then has to return some $F(x')$, where x' might be different than x. This makes sense, because lock_exc is

supposed to be a write-lock, so the client should be able to manipulate the given resource at will, provided it restores the lock's invariants.

Acquiring the shared lock is more interesting, since we have to acquire some $F(x)$ resource in a *shared* way. This is where the $\twoheadrightarrow$ operator comes in: rather than receiving $F(x)$ directly, the client obtains a special resource $\mathsf{Sh}(\gamma, x)$ (for some x), for which we have $\mathsf{Sh}(\gamma, x) \twoheadrightarrow F(x)$. Thus, the client has shared access to $F(x)$ as long as it has the Sh, which must be relinquished upon release of the lock.

From this specification, we can already see the shape of how Leaf will relate back to shared state in Verus. Recall Rust's `RwLock` specification (§3.6.4) and compare it to the spec in Figure 4.6:

- `RwLock::new` returns ownership of `RwLock<T>`, while rwlock_new produces exclusively owned proposition IsRwLock.
- `RwLock::write` and `RwLock::read` require a shared reference to the lock (`&self`), while lock_exc and lock_shared require guard-shared IsRwLock.
- `RwLock::write` returns an object (`RwLockWriteGuard`) that gives full mutable access to the underlying `T`, while lock_exc returns ownership of $F(x)$.
- `RwLock::read` returns an object (`RwLockReadGuard`) that gives shared access to the underlying `T`, while lock_shared returns a proposition ($\mathsf{Sh}(\gamma, x)$) that represents guard-shared access to the $F(x)$.
- `RwLock::into_inner` consumes ownership the lock and returns ownership of the `T`, while rwlock_free consumes ownership of the IsRwLock proposition and returns ownership of the $F(x)$ proposition.

Furthermore, we can start to see how we are going to answer the question asked at the end of §4.4 and formalize the idea of shared references to ghost state. Specifically, a shared reference to ghost state can be understood as *something* $\twoheadrightarrow \boxed{x}^{\gamma}$. Exactly what that "something" is we will return to in Chapter 6.

4.5.3 Elementary deduction rules for the guards operator

Figure 4.7 shows the core deduction rules about the guards operator. It is necessary to point out now that the guards operator, like the view shift, uses *masks*, denoted like $G \twoheadrightarrow_{\mathcal{E}} P$. In Leaf, the mask $\mathcal{E}$ can be interpreted as something like "the set of invariant names that must be openable in order to apply $G \twoheadrightarrow_{\mathcal{E}} P$."

Introduction rules for $\twoheadrightarrow$

The easiest way to introduce the $\twoheadrightarrow$ operator is to use Guard-Forever, which takes a P and makes it shareable forever, represented by $\mathsf{True} \twoheadrightarrow_{\mathcal{N}} \triangleright P$. This effectively says, "when True, we have P shared," but of course, True is always true, so we just have P shared forever. This is a lot like allocating an invariant $\boxed{P}^{\mathcal{N}}$ as would be done in traditional Iris; in fact, we can use $\mathsf{True} \twoheadrightarrow_{\mathcal{N}} P$ to get $\boxed{P}^{\mathcal{N}}$ (Guard-Invariant), though not the other way around.[2]

[2]The reason why $\mathsf{True} \twoheadrightarrow_{\mathcal{N}} P$ is stronger than $\boxed{P}^{\mathcal{N}}$ has to do with Iris's definition of an invariant in terms of the update modality. In order for Leaf's $\wedge$-related rules to work out, our construction of the $\twoheadrightarrow$ operator cannot use

Deduction rules for guarded resources

Persistent Propositions: $P \twoheadrightarrow_{\mathcal{E}} Q$ (where $P, Q : \mathit{iProp}$, $\mathcal{E} : \mathscr{P}(\mathit{Name})$)

GUARD-FOREVER
$$\frac{\mathcal{N} \text{ infinite}}{P \Rrightarrow_{\mathcal{N}} (\mathsf{True} \twoheadrightarrow_{\mathcal{N}} \triangleright P)}$$

GUARD-INVARIANT
$$(\mathsf{True} \twoheadrightarrow_{\mathcal{E}} P) \vdash \boxed{P}^{\mathcal{E}}$$

GUARD-UPD
$$\frac{\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset}{(G \twoheadrightarrow_{\mathcal{E}_1} P) * (P * A \Rrightarrow_{\mathcal{E}_2} P * B) \vdash (G * A \Rrightarrow_{\mathcal{E}_1 \cup \mathcal{E}_2} G * B)}$$

GUARD-OPEN
$$(P \twoheadrightarrow_{\mathcal{E}} Q) \vdash P \ {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} Q * (Q \ {}^{\emptyset}\!\Rrightarrow\!\!\!\!*^{\mathcal{E}} P)$$

GUARD-REFL
$$P \twoheadrightarrow_{\mathcal{E}} P$$

GUARD-TRANS
$$(P \twoheadrightarrow_{\mathcal{E}} Q) * (Q \twoheadrightarrow_{\mathcal{E}} R) \vdash (P \twoheadrightarrow_{\mathcal{E}} R)$$

GUARD-SPLIT
$$P * Q \twoheadrightarrow_{\mathcal{E}} P$$

GUARD-WEAKEN-MASK
$$(G \twoheadrightarrow_{\mathcal{E}_1} P) \vdash (G \twoheadrightarrow_{\mathcal{E}_1 \cup \mathcal{E}_2} P)$$

GUARD-PERS
$$\frac{\mathsf{persistent}(C)}{(G \twoheadrightarrow_{\mathcal{E}} A) * C \vdash (G \twoheadrightarrow_{\mathcal{E}} A * C)}$$

UNGUARD-PERS
$$\frac{A * B \vdash C \quad \mathsf{persistent}(C)}{G * (G \twoheadrightarrow_{\mathcal{E}} A) * B \Rrightarrow_{\mathcal{E}} G * (G \twoheadrightarrow_{\mathcal{E}} A) * B * C}$$

POINTPROP-OWN
$$\mathsf{point}(\boxed{x}^{\gamma})$$

GUARD-IMPLIES
$$\frac{A \vdash P \quad \mathsf{point}(P)}{(G \twoheadrightarrow_{\mathcal{E}} A) \vdash (G \twoheadrightarrow_{\mathcal{E}} P)}$$

POINTPROP-SEP
$$\frac{\mathsf{point}(P) \quad \mathsf{point}(Q)}{\mathsf{point}(P * Q)}$$

GUARD-AND
$$\frac{A \wedge B \vdash P \quad \mathsf{point}(P)}{(G \twoheadrightarrow_{\mathcal{E}} A) * (G \twoheadrightarrow_{\mathcal{E}} B) \vdash (G \twoheadrightarrow_{\mathcal{E}} P)}$$

GUARD-OR-CANCEL
$$\frac{A \wedge C \vdash \mathsf{False}}{(A \twoheadrightarrow_{\mathcal{E}} (B \vee C)) \vdash (A \twoheadrightarrow_{\mathcal{E}} B)}$$

GUARD-OR-CANCEL-G
$$\frac{C \twoheadrightarrow_{\mathcal{E}} \mathsf{False}}{(A \twoheadrightarrow_{\mathcal{E}} (B \vee C)) \vdash (A \twoheadrightarrow_{\mathcal{E}} B)}$$

Figure 4.7: **Deduction rules for $\twoheadrightarrow$.**

Deduction rules for the guards-with-laters

Persistent Propositions: $P \twoheadrightarrow^{\triangleright n}_{\mathcal{E}} Q$ (where $P, Q : \mathit{iProp}$, $\mathcal{E} : \mathscr{P}(\mathit{Name})$, $n \in \mathbb{N}$)

$$P \twoheadrightarrow^{\triangleright 0}_{\mathcal{E}} Q \dashv\vdash P \twoheadrightarrow_{\mathcal{E}} Q$$

Guard-Weaken-Later
$$\frac{n \leq m}{P \twoheadrightarrow^{\triangleright n}_{\mathcal{E}} Q \vdash P \twoheadrightarrow^{\triangleright m}_{\mathcal{E}} Q}$$

Guard-Later-Absorb
$$(P \twoheadrightarrow^{\triangleright n}_{\mathcal{E}} \triangleright Q) \vdash (P \twoheadrightarrow^{\triangleright n+1}_{\mathcal{E}} Q)$$

Guard-Later-Open
$$(P \twoheadrightarrow^{\triangleright n}_{\mathcal{E}} Q) \vdash P \ {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \ \triangleright^n \Rrightarrow_{\emptyset} Q * (Q \ {}^{\emptyset}\!\Rrightarrow\!\!\!*^{\mathcal{E}} P)$$

Figure 4.8: **Selected deduction rules for guards-with-laters.**

Applying $\twoheadrightarrow$

Capturing the intuition that $G \twoheadrightarrow P$ means that "G stands in for a read-only P," Guard-Upd shows that if we have some view shift which leaves P unchanged, then we can make that update with G instead. Guard-Open is a little more flexible than Guard-Upd, allowing us to perform a view shift to obtain P, then view shift back at our convenience. This is similar to Inv-Open.

Observe that Guard-Upd allows us to make use of $(a \cdot x) \rightsquigarrow (b \cdot x)$, which was one of our challenges from earlier. Specifically, we can start with RA-Update:

$$\boxed{a}^{\gamma} * \boxed{x}^{\gamma} \Rrightarrow \boxed{b}^{\gamma} * \boxed{x}^{\gamma}$$

Then if we have $G \twoheadrightarrow_{\mathcal{E}} \boxed{x}^{\gamma}$, we can apply Guard-Upd to get:

$$\boxed{a}^{\gamma} * G \Rrightarrow \boxed{b}^{\gamma} * G$$

The logic of $\twoheadrightarrow$

The remaining rules show additional ways we can manipulate and compose $\twoheadrightarrow$ operations. Guarding is reflexive (Guard-Refl) and transitive (Guard-Trans).

One might hope for a rule that lets us compose $G \twoheadrightarrow A$ and $A \vdash P$ to get $G \twoheadrightarrow P$. Unfortunately this rule is not sound, though it does hold for some special cases. Specifically, Guard-Implies shows that this holds if we have a certain technical condition on P: If P is of the form $\boxed{x}^{\gamma}$, for example, which is a common case, then this condition holds and we can apply Guard-Implies.

Guard-And shows that we can combine two $\twoheadrightarrow$ relationships by applying the non-separating conjunction $\wedge$, though this likewise requires the same technical condition. Guard-Or-Cancel shows how we can reduce $A \twoheadrightarrow (B \vee C)$ to $A \twoheadrightarrow B$ if we can deductively rule out C.

4.5.4 Using $\twoheadrightarrow$ to construct read-write permissions

Let us see how to use Leaf to construct a read-write permission system. Here, we will do counting permissions again, roughly similar to what we did in §3.6.3.

any update modalities internally. In some sense, True $\twoheadrightarrow_{\mathcal{N}} P$ says "We can open the invariants to get P *without having to perform any updates*."

First of all, we can construct ghost state (using the basic RA rules) with the following properties:

$$\mathsf{True} \Rrightarrow \exists \gamma.\ \boxed{\mathsf{Reading}}^{\gamma} * \boxed{\mathsf{Counter}(0)}^{\gamma}$$

$$\boxed{\mathsf{Counter}(n)}^{\gamma} \Lrightarrow \boxed{\mathsf{Counter}(n+1)}^{\gamma} * \boxed{\mathsf{ReadRef}}^{\gamma}$$

$$\boxed{\mathsf{Counter}(0)}^{\gamma} * \boxed{\mathsf{Reading}}^{\gamma} \Lrightarrow \boxed{\mathsf{NoCounter}}^{\gamma} * \boxed{\mathsf{Writing}}^{\gamma}$$

$$\boxed{\mathsf{ReadRef}}^{\gamma} \wedge \boxed{\mathsf{Writing}}^{\gamma} \vdash \mathsf{False}$$

Now, using Guard-Forever, we can say, for an arbitrary proposition P:

$$P \Rrightarrow \exists \gamma.\ \left(\mathsf{True} \twoheadrightarrow_{\mathcal{N}} \triangleright(\boxed{\mathsf{Writing}}^{\gamma} \vee (\boxed{\mathsf{Reading}}^{\gamma} * P))\right) * \boxed{\mathsf{Counter}(0)}^{\gamma}$$

To do this, we initialize the ghost state with $\boxed{\mathsf{Reading}}^{\gamma} * \boxed{\mathsf{Counter}(0)}^{\gamma}$, then put $\boxed{\mathsf{Reading}}^{\gamma} * P$ inside the guard, while holding onto $\boxed{\mathsf{Counter}(0)}^{\gamma}$.

Now, by applying Guard-Open (or by using Guard-Invariant and applying standard invariant rules), we can say:

$$\boxed{\mathsf{Counter}(0)}^{\gamma} \Rrightarrow_{\mathcal{N}} \boxed{\mathsf{Writing}}^{\gamma} * \triangleright P$$

and the reverse:

$$\boxed{\mathsf{Writing}}^{\gamma} * \triangleright P \Rrightarrow_{\mathcal{N}} \boxed{\mathsf{Counter}(0)}^{\gamma}$$

The only thing left to do is show that $\boxed{\mathsf{ReadRef}}^{\gamma}$ propositions are readers. Specifically, we want something like $\boxed{\mathsf{ReadRef}}^{\gamma} \twoheadrightarrow_{\mathcal{N}} \triangleright P$.

To start, we can trivially construct $\boxed{\mathsf{ReadRef}}^{\gamma} \twoheadrightarrow \mathsf{True}$, and then by Guard-Trans:

$$\boxed{\mathsf{ReadRef}}^{\gamma} \twoheadrightarrow_{\mathcal{N}} \triangleright(\boxed{\mathsf{Writing}}^{\gamma} \vee (\boxed{\mathsf{Reading}}^{\gamma} * P))$$

By Guard-Or-Cancel and the fact that $\boxed{\mathsf{ReadRef}}^{\gamma} \wedge \boxed{\mathsf{Writing}}^{\gamma} \vdash \mathsf{False}$, we get:

$$\boxed{\mathsf{ReadRef}}^{\gamma} \twoheadrightarrow_{\mathcal{N}} \triangleright(\boxed{\mathsf{Reading}}^{\gamma} * P)$$

Which we can simplify to:

$$\boxed{\mathsf{ReadRef}}^{\gamma} \twoheadrightarrow_{\mathcal{N}} \triangleright P$$

Bringing it all together, we have our counting permissions:

$$\boxed{\mathsf{NoCounter}}^{\gamma} * \triangleright P \Lrightarrow_{\mathcal{N}} \boxed{\mathsf{Counter}(0)}^{\gamma}$$
$$\boxed{\mathsf{Counter}(n)}^{\gamma} \Lrightarrow \boxed{\mathsf{Counter}(n+1)}^{\gamma} * \boxed{\mathsf{ReadRef}}^{\gamma}$$
$$\boxed{\mathsf{ReadRef}}^{\gamma} \twoheadrightarrow_{\mathcal{N}} \triangleright P$$

Observe that this is the **deposit/withdraw/guard pattern** again, just expressed in a different language. We can *deposit* a P (the first rule, forward direction), *withdraw* a P (the first rule, backward direction), or *guard* (the last rule).

Resource algebras			Storage protocols		
Carrier monoid M			Protocol monoid P Storage monoid S		
Validity $\mathcal{V} : M \to \mathit{Prop}$			Relation $\mathcal{R} : P \times S \to \mathit{Prop}$		
Derived relations			Derived relations		
$\rightsquigarrow$	To derive $\Rrightarrow$	(RA-Update)	$\rightsquigarrow$	To derive $\Rrightarrow$	(SP-Update)
			$\dot{\rightsquigarrow}$	To derive deposits $\Rrightarrow$	(SP-Deposit)
			$\rightsquigarrow\!\!\!\!\cdot$	To derive withdrawals $\Rrightarrow$	(SP-Withdraw)
			$\dot{\rightsquigarrow}\!\!\!\!\cdot$	To derive exchanges $\Rrightarrow$	(SP-Exchange)
			$\twoheadrightarrow$	To derive guards $\twoheadrightarrow$	(SP-Guard)

Table 4.1: **Comparison of resource algebras and storage protocols.**

4.5.5 Storage protocols

In this section, we devise a general form for the above kind of protocol, something to systematize the deposit/withdraw/guard pattern, and which has the 'shape of an RA' such that it can be translated into Verus in a similar way as we saw previously.

This general form we devise is called a *storage protocol*. As in a resource algebra, the storage protocol uses the elements of monoid to define resources. As in a resource algebra, the storage protocol defines a relation ($\rightsquigarrow$) which can be used to derive updates ($\Rrightarrow$).

However, the storage protocol has some additional features. The storage protocol defines a *deposits* relation ($\dot{\rightsquigarrow}$) which can be used to define a view shift ($\Rrightarrow$) that deposits a proposition. It also defines a *withdraws* relation ($\rightsquigarrow\!\!\!\!\cdot$) which can be used to define a view shift ($\Rrightarrow$) that withdraws a proposition. Most generally, it defines an *exchanges* relation ($\dot{\rightsquigarrow}\!\!\!\!\cdot$) which can do both. (Thus, $\rightsquigarrow$, $\dot{\rightsquigarrow}$, and $\rightsquigarrow\!\!\!\!\cdot$ are all just special cases of $\dot{\rightsquigarrow}\!\!\!\!\cdot$.)

Finally, it defines a relation $\twoheadrightarrow$ which can be used to derive $\twoheadrightarrow$ propositions. All the relationships are summarized in Table 4.1, and the precise definitions are given in Figure 4.9b.

How exactly do we determine what propositions are withdrawn or stored? I wanted to avoid defining a storage protocol in terms of *iProp*, as that would make it a recursive construction and force it to explicitly deal with step-indexing. In order to avoid this, I defined the storage protocol with respect to a *storage monoid* S. All the well-formedness conditions and derived relations ($\dot{\rightsquigarrow}\!\!\!\!\cdot$, $\twoheadrightarrow$) are defined in terms of S. Then, when the user instantiates the protocol in the logic (e.g., in SP-Alloc), they supply a function $F : S \to \mathit{iProp}$.

The carrier monoid, also called the *protocol monoid*, for the storage protocol is denoted P, and the storage protocol denoted S. They are related by an arbitrary relation $\mathcal{R}$.[3] The usual validity predicate $\mathcal{V}$ is omitted for P; validity is already implicit in $\mathcal{R}$ (SP-Valid). The relation $\mathcal{R}$ effectively describes "the valid storage states for a given protocol state." Observe that the definition $\dot{\rightsquigarrow}\!\!\!\!\cdot$ requires us to preserve $\mathcal{R}$. Also inspect the definition of $\twoheadrightarrow$, which lets us derive $\twoheadrightarrow$: $p \twoheadrightarrow s$ effectively says, "for any valid protocol state containing p, any corresponding storage state must contain s."

[3]This is a bit different than the Leaf paper, which uses a function $P \to S$. The relation $\mathcal{R}$ is a bit more general, and in fact, we will need it in Chapter 5.

A **storage protocol** consists of:
A *storage monoid*, that is, a partial commutative monoid $(S, \cdot, \mathcal{V})$, where,

$$\begin{aligned}
&\forall a.\ a \cdot \epsilon = a \\
&\forall a, b.\ a \cdot b = b \cdot a \\
&\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \\
&\mathcal{V}(\epsilon) \\
&\forall a, b.\ a \preceq b \land \mathcal{V}(b) \Rrightarrow \mathcal{V}(a)
\end{aligned}$$

A *protocol monoid*, that is, a (total) commutative monoid $(P, \cdot)$, with an arbitrary relation $\mathcal{R} : P \times S \to \mathit{Prop}$ where,

$$\begin{aligned}
&\forall a.\ a \cdot \epsilon = a \\
&\forall a, b.\ a \cdot b = b \cdot a \\
&\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \\
&\forall a, s.\ \mathcal{R}(a, s) \Rrightarrow \mathcal{V}(s)
\end{aligned}$$

Note that $\mathcal{R}$ (unlike $\mathcal{V}$) is *not* necessarily closed under $\preceq$.
Derived relations for storage protocols:

For $p, p' : P$ and $s, s' : S$, define:

$$\begin{aligned}
(p, s) \rightsquigarrow\!\!\!\!\cdot\ (p', s') &\triangleq \forall q, t.\ \mathcal{R}(p \cdot q, t) \Rrightarrow \\
&\quad \exists t'.\ \mathcal{R}(p' \cdot q, t') \\
&\quad \land\ \mathcal{V}(t \cdot s) \\
&\quad \land\ t \cdot s = t' \cdot s' \\
p \rightsquigarrow\!\!\!\!\cdot\ (p', s') &\triangleq (p, \epsilon) \rightsquigarrow\!\!\!\!\cdot\ (p', s') \\
(p, s) \rightsquigarrow p' &\triangleq (p, s) \rightsquigarrow\!\!\!\!\cdot\ (p', \epsilon) \\
p \rightsquigarrow p' &\triangleq (p, \epsilon) \rightsquigarrow\!\!\!\!\cdot\ (p', \epsilon) \\
p \twoheadrightarrow s &\triangleq \forall q, t.\ \mathcal{R}(p \cdot q, t) \Rrightarrow s \preceq t
\end{aligned}$$

(a) Definitions.

Storage Protocol Logic
Instantiated for a given storage protocol $(S, \cdot, \mathcal{V}), (P, \cdot), \mathcal{R}$
Propositions: $\langle p \rangle^\gamma$
Persistent propositions: $\mathrm{sto}(\gamma, F)$
(where $\gamma : \mathit{Name}$, $F : S \to \mathit{iProp}$, $p : P$)

$\mathrm{RespectsComposition}(F) \triangleq (F(\epsilon) \dashv\vdash \mathsf{True})$
and $\forall x, y.\ \mathcal{V}(x \cdot y) \Rrightarrow (F(x \cdot y) \dashv\vdash F(x) * F(y))$

SP-Alloc
$$\frac{\mathrm{RespectsComposition}(F) \quad \mathcal{R}(p, s) \quad \mathcal{N}\ \text{infinite}}{F(s) \Rrightarrow \exists \gamma.\ \mathrm{sto}(\gamma, F) * \langle p \rangle^\gamma * (\gamma \in \mathcal{N})}$$

SP-Exchange
$$\frac{(p, s) \rightsquigarrow\!\!\!\!\cdot\ (p', s')}{\mathrm{sto}(\gamma, F) \vdash (\triangleright F(s)) * \langle p \rangle^\gamma \Rrightarrow_\gamma (\triangleright F(s')) * \langle p' \rangle^\gamma}$$

SP-Deposit
$$\frac{(p, s) \rightsquigarrow p'}{\mathrm{sto}(\gamma, F) \vdash (\triangleright F(s)) * \langle p \rangle^\gamma \Rrightarrow_\gamma \langle p' \rangle^\gamma}$$

SP-Withdraw
$$\frac{p \rightsquigarrow\!\!\!\!\cdot\ (p', s')}{\mathrm{sto}(\gamma, F) \vdash \langle p \rangle^\gamma \Rrightarrow_\gamma (\triangleright F(s')) * \langle p' \rangle^\gamma}$$

SP-Update
$$\frac{p \rightsquigarrow p'}{\mathrm{sto}(\gamma, F) \vdash \langle p \rangle^\gamma \Rrightarrow_\gamma \langle p' \rangle^\gamma}$$

SP-PointProp
$\mathsf{point}(\langle p \rangle^\gamma)$

SP-Guard
$$\frac{p \twoheadrightarrow s}{\mathrm{sto}(\gamma, F) \vdash \langle p \rangle^\gamma \twoheadrightarrow_\gamma (\triangleright F(s))}$$

SP-Unit
$\mathrm{sto}(\gamma, F) \vdash \langle \epsilon \rangle^\gamma$

SP-Sep
$\langle p \cdot q \rangle^\gamma \dashv\vdash \langle p \rangle^\gamma * \langle q \rangle^\gamma$

SP-Valid
$\langle p \rangle^\gamma \vdash \exists q, t.\ \mathcal{R}(p \cdot q, t)$

SP-And
$\langle a \rangle^\gamma \land \langle b \rangle^\gamma \vdash \exists c.\ \langle c \rangle^\gamma * (a \preccurlyeq c) * (b \preccurlyeq c)$

(b) Deduction rules.

Figure 4.9: **Storage protocols, derived relations, and deduction rules.**

$$
\begin{aligned}
(p, s) \dot{\rightsquigarrow} Z \triangleq \forall q, t.\, \mathcal{R}(p \cdot q, t) \Rightarrow \exists p', s', t'.\, (' p, s') \in Z \\
\wedge\, \mathcal{R}(p' \cdot q, t') \\
\wedge\, \mathcal{V}(t \cdot s) \\
\wedge\, t \cdot s = t' \cdot s'
\end{aligned}
$$

SP-Exchange-Nondeterministic

$$
\frac{(p, s) \dot{\rightsquigarrow} Z}{\mathrm{sto}(\gamma, F) \vdash (\triangleright\, F(s)) * \langle p \rangle^{\gamma} \Rrightarrow_{\gamma} \exists p', s'.\, ((p', s') \in Z) * (\triangleright\, F(s')) * \langle p' \rangle^{\gamma}}
$$

SP-Exchange-Guarded

$$
\frac{(p \cdot x, s) \dot{\rightsquigarrow} (p' \cdot x, s') \quad \gamma \in \mathcal{E}}{\mathrm{sto}(\gamma, F) * (G \twoheadrightarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}} \langle x \rangle^{\gamma}) \vdash (\triangleright\, F(s)) * \langle p \rangle^{\gamma} * G \Rrightarrow_{\mathcal{E}} (\triangleright\, F(s')) * \langle p' \rangle^{\gamma} * G}
$$

SP-Exchange-Guarded-Nondeterministic

$$
\frac{(p \cdot x, s) \dot{\rightsquigarrow} \{(p' \cdot x, s') \mid (p', s') \in Z\} \quad \gamma \in \mathcal{E}}{\begin{array}{l}\mathrm{sto}(\gamma, F) * (G \twoheadrightarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}} \langle x \rangle^{\gamma}) \vdash (\triangleright\, F(s)) * \langle p \rangle^{\gamma} * G \\ \qquad \Rrightarrow_{\mathcal{E}} \exists p', s'.\, ((p', s') \in Z) * (\triangleright\, F(s')) * \langle p' \rangle^{\gamma} * G\end{array}}
$$

SP-Exchange-Guarded-Nondeterministic-Later

$$
\frac{(p \cdot x, s) \dot{\rightsquigarrow} \{(p' \cdot x, s') \mid (p', s') \in Z\} \quad \gamma \in \mathcal{E}}{\begin{array}{l}\mathrm{sto}(\gamma, F) * (G \twoheadrightarrow\!\!\!\!\twoheadrightarrow_{\mathcal{E}}^{\triangleright n} \langle x \rangle^{\gamma}) \vdash (\triangleright\, F(s)) * \langle p \rangle^{\gamma} * G \\ \qquad \Rrightarrow_{\mathcal{E}, \emptyset}\ \triangleright^{n} \Rrightarrow_{\emptyset, \mathcal{E}} \exists p', s'.\, ((p', s') \in Z) * (\triangleright\, F(s')) * \langle p' \rangle^{\gamma} * G\end{array}}
$$

Figure 4.10: **"Advanced" rules for storage protocols.**

4.5.6 More advanced rules

This section covers some Leaf rules which are not strictly needed to understand the core Leaf concepts, but which will be technically necessary in Chapter 6.

Guards that involve laters

As with invariants, the later modality ($\triangleright$) often shows up in $\twoheadrightarrow$ propositions, e.g., in SP-Guard. The Leaf paper provides a rule for eliminating these $\triangleright$ modalities in some cases; however, that rule is not always applicable, and sometimes we need something more general.

For this purpose, we define the *guards-with-laters* operator $P \twoheadrightarrow^{\triangleright n}_{\mathcal{E}} Q$, which is basically like $P \twoheadrightarrow_{\mathcal{E}} Q$ except it requires us to take n later-steps when applied. Figure 4.8 shows some laws for manipulating the guards-with-laters.

The most general "exchange" rule

The SP-Exchange needs to be made more general in a few dimensions.

- First, we have SP-Exchange-Nondeterministic, a nondeterministic version of SP-Exchange, in the same way that RA-Update-Nondeterministic is a nondeterministic version of RA-Update. Rather than specifying a single value that they want to transition to, the user specifies a set of potential values in the set Z.
- Next, we have SP-Exchange-Guarded. At first glance, this appears to follow from our existing rules. One might reason: If we have $(p \cdot x, s) \rightsquigarrow (p' \cdot x, s')$, then we can apply SP-Exchange to get a view shift with $\langle x \rangle^\gamma$ on both sides. Then we can apply Guard-Upd. However, this train of logic requires a *disjointness condition* on the masks from the hypothesis of Guard-Upd.

  It turns out this disjointness condition is unnecessary, as demonstrated by SP-Exchange-Guarded, which has the name γ in both relevant masks. The proof of SP-Exchange-Guarded is quite subtle; effectively, we have to "open two possibly-overlapping invariants simultaneously," which involves careful use of the non-separating conjunction $\wedge$.

Now that we have two described improvements, we can combine them into one rule that handles both: SP-Exchange-Guarded-Nondeterministic.

Finally, we can write the version for $\twoheadrightarrow^{\triangleright n}_{\mathcal{E}}$, bringing us to our Most General Exchange Rule: SP-Exchange-Guarded-Nondeterministic-Later.

4.6 Leaf Storage Protocols in Verus

We can translate the storage protocols laws into a Verus interface in much the same way we did for resource algebras. Figure 4.11 shows the definition of a storage protocol as a trait. Figure 4.12 shows some of the basics, similar to what we had for resource algebras, and Figure 4.13 has the special storage protocol operations: guarding and exchanging.

There are a few things worth calling out.

```
pub trait Protocol<K, V>: Sized {
    // Definition of a Storage Protocol

    spec fn op(self, other: Self) -> Self;   // ·
    spec fn rel(self, s: Map<K, V>) -> bool; // ℛ
    spec fn unit() -> Self;                  // ϵ

    // Well-formedness conditions for a storage protocol (Figure 4.9a)

    proof fn commutative(a: Self, b: Self)
        ensures Self::op(a, b) == Self::op(b, a);

    proof fn associative(a: Self, b: Self, c: Self)
        ensures Self::op(a, Self::op(b, c)) == Self::op(Self::op(a, b), c);

    proof fn op_unit(a: Self)
        ensures Self::op(a, Self::unit()) == a;
}

// ≼
pub open spec fn incl<K, V, P: Protocol<K, V>>(a: P, b: P) -> bool {
    exists|c| P::op(a, c) == b
}
```

Figure 4.11: **Verus's ghost state encoding of a Storage Protocol (Part I).**

The storage monoid The Leaf storage protocols are written with respect to an arbitrary "storage monoid." In practice, it usually seems good enough to make it some kind of map type. In any case, we need something that works well in Verus, and it happens that maps work well since `Map<K, V>` is the primary Verus type for arbitrary collections of ghost objects. Here, `V` is the type of the ghost object we want to store, and `K` is some arbitrary index type.

Thread safety and the Send/Sync traits We need to be somewhat careful about the `Send` and `Sync` marker traits (§3.4.6). Observe that if `StorageResource<K, V, P>` is sent or synced to another thread then it may be possible (via withdrawing or guarding) for a `V` to be sent or synced as well. Thus, the `Send` and `Sync` traits on `StorageResource` are conditional on those of `V`:

```
impl<V : Send + Sync> Send for StorageResource<K, V, P>
impl<V : Send + Sync> Sync for StorageResource<K, V, P>
```

It is possible that this is overly conservative for some protocols, e.g., some protocols might not support any guarding operations at all, and in such a case `StorageResource<K, V, P>` could be marked `Sync` regardless. Currently, however, any such structure is not taken into account.

4.7 Recap

We explored ghost state theories in the Iris separation logic. Using the standard ghost state formalism of the *Resource Algebra*, we were able to motivate the definition of a ghost state interface in Verus. We also developed a new ghost state formalism, the *Storage Protocol*, which

```
// Ghost state representing ⟨p⟩^γ
pub tracked type StorageResource<K, V, P>

impl<K, V, P: Protocol<K, V>> StorageResource<K, V, P> {
    // Spec encoding of a StorageResource
    pub open spec fn value(self) -> P; // p
    pub open spec fn loc(self) -> Loc; // γ

    // SP-Alloc
    pub proof fn alloc(p: P, tracked s: Map<K, V>) -> (tracked out: Self)
        requires P::rel(p, s),
        ensures out.value() == p;

    // SP-Sep (going backward)
    pub proof fn join(tracked a: Self, tracked b: Self) -> (tracked out: Self)
        requires a.loc() == b.loc(),
        ensures
            out.loc() == a.loc(),
            out.value() == P::op(a.value(), b.value());

    // SP-Sep (going forward)
    pub proof fn split(tracked self, a_value: P, b_value: P)
        -> (tracked out: (Self, Self))
        requires self.value() == P::op(a_value, b_value),
        ensures
            out.0.loc() == self.loc(),
            out.1.loc() == self.loc(),
            out.0.value() == a_value,
            out.1.value() == b_value;

    // SP-And
    pub proof fn join_shared<'l>(tracked a: &'l Self, tracked b: &'l Self)
        -> (tracked out: &'l Self)
        requires a.loc() == b.loc(),
        ensures
            out.loc() == a.loc(),
            incl(a.value(), out.value()),
            incl(b.value(), out.value()),

    pub proof fn weaken<'l>(tracked &'l self, target: P) -> (tracked out: &'l Self)
        requires incl(target, self.value()),
        ensures
            out.loc() == self.loc(),
            out.value() == target,

    // SP-Valid
    pub proof fn validate(tracked a: &Self)
        -> (q: P, t: Map<K, V>)
        ensures P::rel(P::op(a.value(), q), t)

    // SP-Valid, but where part is shared
    pub proof fn validate_with_shared(tracked p: &mut Self, tracked x: &Self)
        -> (q: P, t: Map<K, V>)
        requires
            old(p).loc() == x.loc(),
        ensures
            *p == *old(p),
            P::rel(P::op(P::op(p.value(), x.value()), q), t)
}
```

Figure 4.12: **Verus's ghost state encoding of a Storage Protocol (Part II).**

```
// ↠
pub open spec fn guards<K, V, P: Protocol<K, V>>(p: P, b: Map<K, V>) -> bool {
    forall|q: P, t: Map<K, V>| P::rel(P::op(p, q), t) ==> b.submap_of(t)
}

// (p, b) ⇝ Z  (Figure 4.10)
pub open spec fn exchanges_nondeterministic<K, V, P: Protocol<K, V>>(
    p1: P,
    s1: Map<K, V>,
    new_values: Set<(P, Map<K, V>)>,
) -> bool {
    forall |q: P, t1: Map<K, V>|
        P::rel(P::op(p1, q), t1) ==> exists |p2: P, s2: Map<K, V>, t2: Map<K, V>|
            new_values.contains((p2, s2))
              && P::rel(P::op(p2, q), t2)
              && t1.dom().disjoint(s1.dom())
              && t2.dom().disjoint(s2.dom())
              && t1.union_prefer_right(s1) =~= t2.union_prefer_right(s2)
}

// {(q · t, s) | (q, s) ∈ set}
pub open spec fn set_op<K, V, P: Protocol<K, V>>(set: Set<(P, Map<K, V>)>, t: P)
    -> Set<(P, Map<K, V>)>
{
    Set::new(|v: (P, Map<K, V>)| exists |q| set.contains((q, v.1)) && v.0 == P::op(q, t))
}

pub trait Protocol<K, V>: Sized {
impl<K, V, P: Protocol<K, V>> StorageResource<K, V, P> {
    // SP-Guard
    pub proof fn guard<'l>(tracked p: &'l Self, s_value: Map<K, V>)
        -> (tracked s: &'l Map<K, V>)
        requires
            guards(p.value(), s_value),
        ensures
            s == s_value;

    // SP-Exchange-Guarded-Nondeterministic
    pub proof fn exchange_nondeterministic_with_shared(
        tracked p: Self,
        tracked x: &Self,
        tracked s: Map<K, V>,
        new_values: Set<(P, Map<K, V>)>,
    ) -> (tracked (new_p: Self, new_s: Map<K, V>))
        requires
            p.loc() == x.loc(),
            exchanges_nondeterministic(
                P::op(p.value(), x.value()),
                s,
                set_op(new_values, x.value()),
            ),
        ensures
            new_p.loc() == p.loc(),
            new_values.contains((new_p.value(), new_s));
}
```

Figure 4.13: **Verus's ghost state encoding of a Storage Protocol (Part III).**

supports the deposit/withdraw/guard pattern, and used it to motivate another ghost state interface in Verus.

In the next chapter, we will see these interfaces as the bases for formalizing a more user-friendly ghost state construction mechanism, VerusSync.

Chapter 5

Ghost State as a Transition System

In this section, we will formally introduce the VerusSync tool for constructing ghost state. To motivate the design of VerusSync, we first need to ask: Why do we want VerusSync at all?

After all, we already have the Verus Monoidal Ghost Interface for constructing ghost state, and it has a lot of advantages. It is based directly on resources algebras, which are a time-tested, general and canonical formalism to represent resources. Storage protocols use similar principles, and this body of work has shown them to at least be general and useful. In fact, the Linear Dafny IronSync framework, which predated VerusSync, used something similar to the Verus Monoidal Ghost Interface. Why, then, is the Verus Monoidal Ghost Interface not enough?

The short answer is something like: "Go look at SP-Exchange-Guarded-Nondeterministic or `exchange_nondeterministic_with_shared`. Do you really want to use that on a regular basis?"

The longer answer is that, from my experience in the IronSync work, it turned out that the compositional structure is an unintuitive lens for thinking about a system. The Resource Algebra definition puts composition front and center: It is the first thing you define, and every proof obligation involves composition in some way. However, when I think about a system, I tend to think of it first in terms of state, transitions, and invariants. In contrast, I think about a system's compositional structure either not at all, or at a subconscious, intuitive level. More concretely, I can say that IronSync's version of the Verus Monoidal Ghost Interface always resulted in a lot of "boilerplate code," and the idea of making a change always instilled a foreboding sense of dread.

The result of all this was the desire to make a system that centered state, transitions, and invariants, which let the composition structure be automatic. Enter VerusSync.

Chapter overview In this chapter, I will first enumerate the features of VerusSync and give some high-level intuition. Then I will define a formal language using a representative subset of features, describe the well-formedness checking process, the token generation process, and finally sketch the proof of soundness.

Strategy	Field has type	Description
Miscellaneous		
`variable`	`V`	One token with the value of the field
`constant`	`V`	Copyable token with the value of the field
`not_tokenized`	`V`	No token
Collections		
`option`	`Option<V>`	One or zero tokens with value `V`
`map`	`Map<K, V>`	One token per `(K, V)` entry
`multiset`	`Multiset<V>`	One token per `V` element
`set`	`Set<V>`	One token per `V` element
`count`	`nat`	Tokens that can add together
`bool`	`bool`	One or zero tokens, no value
Monotonic Collections		
`persistent_option`	`Option<V>`	Copyable tokens with value `V`
`persistent_map`	`Map<K, V>`	Copyable tokens per `(K, V)` entry
`persistent_set`	`Set<V>`	Copyable token per `V` element
`persistent_count`	`nat`	Copyable tokens representing lower-bound
`persistent_bool`	`bool`	Copyable token with no value
Storage		
`storage_option`	`Option<V>`	External token of type `V`, stored or not stored
`storage_map`	`Map<K, V>`	External tokens of type `V`, stored in entries indexed by `K`

Table 5.1: **"Sharding strategies" used by VerusSync.**

5.1 VerusSync Overview

The key idea of VerusSync is that the user writes down a transition system and Verus generates ghost tokens with operations that correspond to the transitions. For this purpose, fields of VerusSync are annotated with *sharding strategies*, which dictate the form that the ghost tokens will take for that field and how they relate to the field's value.

The strategies that are currently implemented in Verus are displayed in Table 5.1. The strategy used in a field also place restrictions on how transitions are able to manipulate that field, providing special commands that need to be used.

Collection strategies For strategies in the "collections" class, tokens can be created and destroyed, e.g., for the `map` strategy, you can add (key, value) pairs to a map or remove them. These are all manipulated with `add` and `remove` commands. One can also use the `have` command to read the value of an entry without modifying it.

Some of these commands are associated with nontrivial proof obligations. For example, if the user tries to `add` a key-value pair to a map—a common operation—this is only sound if we can somehow ensure that the key does not already exist in the map. Otherwise the result would be conflicting entries.

```
transition!{
    t(k: K, v: V) {
        add map_field += [ k => v ]; // ERROR: k might already be in map.
    }
}
```

One might try to just declare this as a precondition like so, but this is invalid:

```
transition!{
    t(k: K, v: V) {
        require !pre.map_field.dom().contains(k); // ERROR: Can't refer directly
                                                  // to `pre.map_field`
        add map_field += [ k => v ];
    }
}
```

Why can't we do this? In short, it is because it needs to be possible to check the preconditions of the transition from the ghost tokens that the user provides when interfacing with the ghost token API. It isn't possible to know the exact value of the map just from the tokens that represent some of its entries.

Thus, satisfying the requirements of `add` is only possible by establishing an invariant on the state that ties the value to some other field. Here is a somewhat contrived example:

```
fields {
    #[sharding(map)] pub a_map: Map<K, V>,
    #[sharding(map)] pub b_map: Map<K, V>,
}

#[invariant]
spec fn a_b_disjoint(self) -> bool {
    self.a_map.dom().disjoint(self.b_map.dom())
}

transition!{
    t(k: K, v: V) {
        remove a_map -= [ k => v ];
        add b_map += [ k => v ];
    }
}
```

This is accepted because the invariant tells us `a_map` and `b_map` have to be disjoint, and since `k` was in the pre-state's `a_map`, it cannot also be in the pre-state's `b_map`.

Remember: **Prove global properties, export local features**. The exported transition just transforms one token into another, but proving the operation is sound requires us to specify an invariant predicate on the global state.

Monotonic Collections A monotonic collection is one where new entries can be added, but never destroyed nor modified. As a result, all the tokens are copyable; they can never become out-of-date (similar to *persistent state* in separation logic).

As a result, you cannot use the `remove` command, and the `add` command works a little differently. If you try to add a `[ k => v ]` pair, for example, the requirement is that it *doesn't disagree* with the existing map. If `(k, v)` is already in the map, there is no issue with creating a new token to represent it, since the token is copyable anyway.

The storage strategies For strategies in the "storage" class, no new tokens are created at all. Instead, the user specifies a preexisting ghost object they want to store. They can be inserted or removed from the system by **deposit** and **withdraw** statements, and it possible to get shared references to them via `guard` statements.

A "bird's eye" view Most transitions are deterministic in the values of the input tokens. However, it is possible to write transitions that are dependent on state not represented in the input tokens. To do this, we use the **`birds_eye`** keyword to get unrestricted access to the `pre` state. A common use-case is when you need to generate a fresh ID:

```
fields {
    #[sharding(map)] pub a_map: Map<K, V>
}
transition!{
    add_new_entry(v: V) {
        birds_eye let k = set_max(pre.a_map.dom()) + 1;
        add a_map += [ k => v ];
    }
}
```

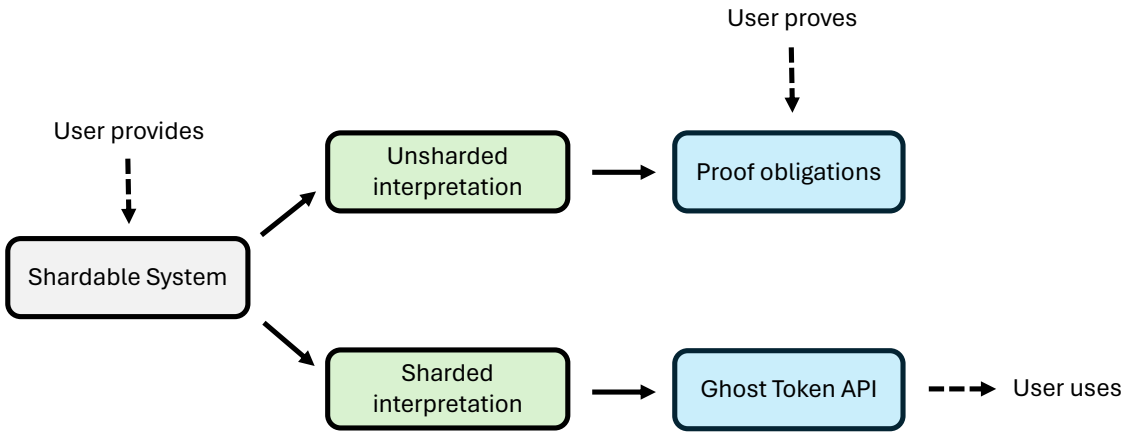


Figure 5.1: **High level picture of VerusSync Core.**

As discussed above, we might not be able to prove *a priori* that some fixed key `k` is absent from the map. However, we can still demonstrate that there exists *some* key which is okay to add. Of course, there are some restrictions on **`birds_eye`**; a precondition cannot depend on **`birds_eye`** data, for example.

The `not_tokenized` strategy The `not_tokenized` strategy is exactly what it sounds like. It doesn't create a token for the given field. Instead, the value is always accessible (via **`birds_eye`**) and always updateable. In most cases, this can be replaced by adding an existential into the state invariant, though sometimes a field is more convenient.

5.2 VerusSync Core

To formalize VerusSync, we will define a formal language we call *VerusSync Core*. VerusSync Core supports a representative subset of VerusSync's sharding strategies. We will informally explain how VerusSync lowers into VerusSync Core, and we will present formal algorithms for verification condition generation and tokenization from VerusSync Core.

The key idea of VerusSync Core is that it lets us define a transition system, which we call the *Shardable System*, in a special way that has *two interpretations*: the *unsharded interpretation*, which allows for the straightforward generation of clean verification conditions, and the *sharded interpretation*, which allows the construction of the ghost token APIs. See Figure 5.1.

5.2.1 The Shardable System

Formally, a Shardable System in VerusSync Core consists of a *ShardableState* and a set of operations.

The shardable state The state *ShardableState* is a list of fields $f_i : (\sigma_i, \tau_i)$ where σ_i is a *sharding strategy*, and τ_i is the type of a field. The core strategies we consider are:

- constant
- variable
- map
- persistent_map
- storage_map

When $\sigma_i \in \{\text{map}, \text{persistent\_map}, \text{storage\_map}\}$, we also require τ_i to be of the form **map** k_i v_i, the type of finite maps from keys k_i to values v_i.

The real implementation has a few additional strategies, but they can be encoded into the core ones. For example,

- $(\text{option}, \textbf{option}\ v)$ can be encoded as $(\text{map}, \textbf{map}\ ()\ v)$
- $(\text{storage\_option}, \textbf{option}\ v)$ can be encoded as $(\text{storage\_map}, \textbf{map}\ ()\ v)$
- $(\text{bool}, \textbf{bool})$ can be encoded as $(\text{map}, \textbf{map}\ ()\ ())$
- $(\text{set}, \textbf{set}\ k)$ can be encoded as $(\text{map}, \textbf{map}\ k\ ())$

(This isn't how VerusSync actually implements these strategies; this is just for illustration's sake.)

The operations A Shardable System comes with a collection of operations, which come in three kinds, *initializers*, *transitions*, and *properties*.

$$\textit{OperationKind} ::= \text{Init} \mid \text{Trans} \mid \text{Property}$$

Each operation is written in a mini-language we call the *Shardable Transition Modeling Language (STML)*. In STML, an operation is given by a sequence of parameters $(p_1, \ldots, p_k)$ and a sequence of *statements* $\textit{stmt}_1, \ldots, \textit{stmt}_m$. The possible statements are given in Figure 5.2. The expressions e may include bound variables $p_1, \ldots, p_k$; non-initializer operations may also include a special variable $\mathbf{s}_{\textit{pre}} : \tau_1 \times \cdots \times \tau_n$.

The statements are divided into several classes, and each operation kind is restricted in the statement classes that it can contain, as detailed at the top of the figure. For example, *InitOp* can only contain *PredicateStmts* and *InitializerStmts*. It cannot contain a *MutatingStmt* or a *ReadonlyStmt* because those kinds of statements interact with the "pre-state" of a transition, whereas there is no "pre-state" in an initializer.

Again, our actual implementation has some additional features which can be interpreted as syntactic sugar over the core statement language. Let-statements, for example, are easily translated by substitution, while let-bindings in statements can be translated by introducing new parameters. VerusSync also has conditionals and match statements, though only in limited cases where the translation remains straightforward. See the examples in Figure 5.3.

Shardable Transition Modeling Language (STML)

Operation types

$$
\begin{aligned}
InitOp &:= [\ \textit{PredicateStmt} \mid \textit{InitializerStmt}\ ]^* && \text{(one initializer per field)} \\
TransOp &:= [\ \textit{PredicateStmt} \mid \textit{MutatingStmt} \mid \textit{ReadonlyStmt}\ ]^* \\
PropertyOp &:= [\ \textit{PredicateStmt} \mid \textit{GuardingStmt} \mid \textit{ReadonlyStmt}\ ]^*
\end{aligned}
$$

Statement types

Predicate statements

require $(e : \textbf{bool})$
assert $(e : \textbf{bool})$

Initializer statements

init $f_i \ \leftarrow \ (e : \tau_i)$

Mutating statements

update f_i **from** $(e_1 : \tau_i)$ **to** $(e_2 : \tau_i)$	for $\sigma_i =$ variable
add f_i += $((e_1 : k_i), (e_2 : v_i))$	for $\sigma_i =$ map
remove f_i -= $((e_1 : k_i), (e_2 : v_i))$	for $\sigma_i =$ map
union $f_i \cup$= $((e_1 : k_i), (e_2 : v_i))$	for $\sigma_i =$ persistent_map
deposit f_i += $((e_1 : k_i), (e_2 : v_i))$	for $\sigma_i =$ storage_map
withdraw f_i -= $((e_1 : k_i), (e_2 : v_i))$	for $\sigma_i =$ storage_map

Read-only statements

be f_i == $(e : \tau_i)$	for $\sigma_i \in \{\text{constant}, \text{variable}\}$
have $f_i \ni ((e_1 : k_i), (e_2 : v_i))$	for $\sigma_i \in \{\text{map}, \text{persistent\_map}\}$

Guarding statements

guard $f_i \ni ((e_1 : k_i), (e_2 : v_i))$	for $\sigma_i =$ storage_map where $\mathbf{s}_{pre}$ is not free in e_1, e_2

Figure 5.2: **The Shardable Transition Modeling Language (STML).**

```
fields {
  #[sharding(variable)] f1: int,
  #[sharding(variable)] f2: int,
}

transition!{
  t(b: bool) {
    if b {
      update f1 = pre.f1 + 5 + pre.f2;
    }
  }
}
```

transition t$(b : \textbf{bool}, pre_1 : \textbf{int}, pre_2 : \textbf{int})$
  be f_2 == pre_2 ;
  update f_1 **from** pre_1 **to** (
    if b **then** $pre_1 + 5 + pre_2$
      else pre_1
)

Example de-sugaring of a transition with a conditional update. The condition is moved into the expression of the **update** statement. In order to refer to the "pre-state" of field `f1`, we add a new parameter, pre_1, and tie it to the value of `f1` through the first argument to **update** statement. Likewise for `f2` with the **be** statement. Generally speaking, we use **be** if a field is read but not written to, and **update** otherwise.

```
fields {
  #[sharding(map)] f1: Map<int, Option<int>>,
}

transition!{
  t(k: int) {
    remove f1 -= [ k => let Some(v) ];
    add f1 += [ k => Some(v + 7) ];
  }
}
```

transition t$(k : \textbf{int}, v : \textbf{int})$
  remove f_1 -= $(k, \text{Some}(v))$;
  add f_1 += $(k, \text{Some}(v + 7))$

Example de-sugaring of a transition with a let-pattern-binding in a `remove` statement. In this case, the bound variable `v` is turned into an additional parameter, v.

```
fields {
    #[sharding(map)] f1: Map<int, int>,
}

transition!{
    t() {
        let birds_eye fresh_key =
            set_max(pre.f1.dom()) + 1;
        add f1 += [ fresh_key => 0 ];
    }
}
```

transition t()
  add f_1 += $(set\_max(\mathbf{s}_{pre}.f_1.dom()) + 1, 0)$

Example de-sugaring of a transition with a **birds_eye** let-binding. The **birds_eye** keyword permits access to the entire pre-state, given by `pre`, allowing the output tokens to not be a deterministic function of the input tokens. In the core STML, this is translated to the special $\mathbf{s}_{pre}$ variable.

Figure 5.3: **Example de-sugarings of the Verus DSL.** *Verus DSL is de-sugared into the core STML (Figure 5.2).*

5.3 The Unsharded Interpretation

Given a Shardable System, we can construct its *unsharded interpretation* and describe what it means for the unsharded interpretation to be well-formed. Further, we also say that the Shardable System is well-formed if its unsharded interpretation is.

To do this, we first define the *unsharded state*, S. Then we show how to translate the initializers, transitions, and properties into a transition system on the state S with a set of *safety conditions*, i.e., conditions that must be shown to hold at every reachable state of the transition system. We then show how to describe well-formedness in terms of finding an *inductive invariant* and checking its associated verification conditions.

Unsharded State Given the fields in the Shardable System, we define the "unsharded state" to be the product of all the field types: $S \triangleq \tau_1 \times \cdots \times \tau_n$.

Translation into an unsharded transition system To turn the given operations into a "transition system," we convert the operations into a simple language with assumes, asserts, and assignments. This language is inspired by *RML* [64], a transition description language used by Ivy, which has been the basis of much work on inductive invariant synthesis and verification, and which is thus appealing for our purposes here. For one, it provides an easy way to specify the verification conditions. Figure 5.4 shows the definition of our SimpleRML, the translation from STML into SimpleRML, and the definition of weakest precondition for SimpleRML.

Well-formed shardable transition systems Formally, we say a VerusSync Core shardable transition system is *well-formed* if there exists an invariant $Inv : S \to \mathbf{bool}$ that satisfies the *verification conditions*, which are defined as, for each operation $\mathbf{op}$,

$$\forall \vec{\mathbf{p}}, \vec{\mathbf{f}}.\ (Inv(\vec{\mathbf{f}}) \Rightarrow wp(\mathbf{op}, Inv(\vec{\mathbf{f}})))[\vec{\mathbf{f}}/\mathbf{s}_{pre}]$$

Here, $\vec{\mathbf{p}}$ is the list of parameters $p_1, \ldots, p_k$ and $\vec{\mathbf{f}}$ is the list of fields $f_1, \ldots, f_n$. The $\mathbf{s}_{pre}$ variable represents the pre-state.

Although it is convenient that we can write the verification conditions in one uniform way across all three operation kinds, one should keep in mind that it behaves a bit differently across the three.

For **transition operations**, the verification condition says that *if* the invariant holds on the pre-state, *then* all of the **assert** statements (safety conditions) will hold, and the invariant will hold on the post-state (the inductiveness criterion).

For **property operations**, the inductiveness criterion is trivial since there can be no field assignment. For properties, the important obligations are all in the **assert** statements, including the ones that arise from translation of **guard** statements.

For **initializer operations**, which are something like "transition operations without a pre-state," we can actually simplify the form of the verification condition to eliminate the appearance of the pre-state variable. Since an initializer is required to have an **init** statement for each field, none of the fields f_i will be free in $wp(\mathbf{op}, Inv(\vec{\mathbf{f}}))$, having been substituted for their initialized

SimpleRML

$$
\begin{aligned}
RML &:= RMLStmt^* \\
RMLStmt &:= \textbf{assume}\ e \mid \textbf{assert}\ e \mid \textbf{assign}\ f_i \leftarrow e
\end{aligned}
$$

STML to SimpleRLM conversion

$$
\begin{aligned}
\textbf{require}\ e &\rightsquigarrow \textbf{assume}\ e \\
\textbf{assert}\ e &\rightsquigarrow \textbf{assert}\ e \\
\\
\textbf{init}\ f_i \leftarrow e &\rightsquigarrow \textbf{assign}\ f_i \leftarrow e \\
\\
\textbf{update}\ f_i\ \textbf{from}\ e_1\ \textbf{to}\ e_2 &\rightsquigarrow \textbf{assume}\ f_i = e_1\ ;\ \textbf{assign}\ f_i \leftarrow e_2 \\
\textbf{add}\ f_i\ \texttt{+=}\ (e_1, e_2) &\rightsquigarrow \textbf{assert}\ e_1 \notin f_i\ ;\ \textbf{assign}\ f_i \leftarrow f_i[e_1 \mapsto e_2] \\
\textbf{remove}\ f_i\ \texttt{-=}\ (e_1, e_2) &\rightsquigarrow \textbf{assume}\ e_1 \in f_i \wedge f_i[e_1] = e_2\ ;\ \textbf{assign}\ f_i \leftarrow f_i \backslash e_1 \\
\textbf{union}\ f_i\ \cup\texttt{=}\ (e_1, e_2) &\rightsquigarrow \textbf{assert}\ e_1 \in f_i \Rightarrow f_i[e_1] = e_2\ ;\ \textbf{assign}\ f_i \leftarrow f_i[e_1 \mapsto e_2] \\
\textbf{deposit}\ f_i\ \texttt{+=}\ (e_1, e_2) &\rightsquigarrow \textbf{assign}\ f_i \leftarrow f_i[e_1 \mapsto e_2] \\
\textbf{withdraw}\ f_i\ \texttt{-=}\ (e_1, e_2) &\rightsquigarrow \textbf{assert}\ e_1 \in f_i \wedge f_i[e_1] = e_2\ ;\ \textbf{assign}\ f_i \leftarrow f_i \backslash e_1 \\
\\
\textbf{be}\ f_i\ \texttt{==}\ e &\rightsquigarrow \textbf{assume}\ f_i = e \\
\textbf{have}\ f_i \ni (e_1, e_2) &\rightsquigarrow \textbf{assume}\ e_1 \in f_i \wedge f_i[e_1] = e_2 \\
\\
\textbf{guard}\ f_i \ni (e_1, e_2) &\rightsquigarrow \textbf{assert}\ e_1 \in f_i \wedge f_i[e_1] = e_2
\end{aligned}
$$

Weakest Precondition

$$
\begin{aligned}
wp(\textbf{assume}\ e\ ;\ stmts, \varphi) &\triangleq e \Rightarrow wp(stmts, \varphi) \\
wp(\textbf{assert}\ e\ ;\ stmts, \varphi) &\triangleq e \wedge wp(stmts, \varphi) \\
wp(\textbf{assign}\ f_i \leftarrow e\ ;\ stmts, \varphi) &\triangleq wp(stmts, \varphi[e/f_i]) \\
wp(empty, \varphi) &\triangleq \varphi
\end{aligned}
$$

Figure 5.4: **SimpleRML.**

values. Also recall that we forbid uses of $\mathbf{s}_{pre}$ is an initializer. As a result, we can use the simpler, equivalent condition,

$$\forall \vec{\mathbf{p}}.\ wp(\mathbf{op}, Inv(\vec{\mathbf{f}}))$$

which says simply that the invariant has to hold on the state after each field has been initialized.

5.4 The Sharded Interpretation

Now, let us turn our attention to the other interpretation of a VerusSync Core System: the *sharded interpretation*.

The sharded interpretation consists of two components: **(i)** a set of token types, and **(ii)** a set of operations on the token types, expressible as functions in the Rust type system, with preconditions and postconditions.

5.4.1 The token types

The collection of token types always consists of a single Instance type and a number of token types derived from the shardable state fields. For each field f_i we get a token type Tok_i (except where σ_i is a storage strategy). Each token type T is assigned a representation type $\lfloor T \rfloor$ (Figure 5.5).

For constant and variable fields, the significant part of the representation is the τ_i field, i.e., the value of the token is the same as the value of the field in the shardable state. For map and persistent_map fields, the significant part of the representation is a key-value pair, i.e., the value of the token represents a single entry in a map.

What about the Instance type? Why do we need it? The main reason has to do with soundness. In general, it's not sound to assume we can perform an operation unless we can ensure that the protocol has already been initialized. Usually, this is automatic: most operations take some input token, and just having the token is a guarantee that the protocol has been initialized. It *is* possible to define a transition that takes 0 input tokens, though. By requiring the Instance token for every operation, we can automatically ensure every operation takes at least 1 token as input.

(In the real VerusSync, Instance plays a few additional roles. First, it serves as a convenient way to talk about the ghost name, γ. We also put all the constant fields on the Instance type, though for simplicity, we do not do this for VerusSync Core, instead putting the constants in their own token types, like the other fields.)

5.4.2 The token operations

Duplication operations The Instance token and any token type Tok_i where,

$$\sigma_i \in \{\mathsf{constant}, \mathsf{persistent\_map}\}$$

is called a *duplicable* token type. For each duplicable token type T, we produce an operation $\mathsf{OpCopy}(T)$:

Per-system token types

$\lfloor \text{Instance} \rfloor \triangleq ()$

Per-field token types

$\lfloor \text{Tok}_i \rfloor \triangleq \tau_i$	for $\sigma_i = \text{constant}$
$\lfloor \text{Tok}_i \rfloor \triangleq \tau_i$	for $\sigma_i = \text{variable}$
$\lfloor \text{Tok}_i \rfloor \triangleq k_i \times v_i$	for $\sigma_i = \text{map}$
$\lfloor \text{Tok}_i \rfloor \triangleq k_i \times v_i$	for $\sigma_i = \text{persistent\_map}$
(No Tok_i)	for $\sigma_i = \text{storage\_map}$

Figure 5.5: **Token types in the sharded interpretation of a VerusSync Core System.**

```
(in: &T) -> (out: T)
    ensures in == out
```

Main operations For each operation t in the VerusSync Core System, we create a token operation as follows: First, using the key in Figure 5.6, map each STML statement to a collection of **inputs**, **outputs**, **preconditions**, and **postconditions**. Then construct a token operation where:

- The input parameters are given by the **inputs**.
  - For Trans and Property operations, also include a single &Instance input parameter.
- The output parameters are given by the **outputs**.
  - For Init operations, also include a single Instance output parameter.
  - If there are any references among the outputs, as is possible for Property operations, they take the same lifetime parameter as the inputs. Also note that for a Property operation, *all* inputs are shared references.
- The precondition is given as the conjunction of the **preconditions**. If any of the preconditions reference $\mathbf{s}_{pre}$, then prepend a $\forall \mathbf{s}_{pre}$.[1] Also add a precondition that all tokens have the same ghost location.
- The postcondition is given as the conjunction of the **postconditions**. If any of the preconditions reference $\mathbf{s}_{pre}$, then add an $\exists \mathbf{s}_{pre}$. Also add a precondition that all output tokens have the same ghost location as the input tokens.

There are several odd aspects to remark on.

- Observe that in the sharded interpretation, the order of the STML statements doesn't matter, even though they *do* matter in the earlier unsharded interpretation. For example, the

[1] Actually, the real VerusSync disallows preconditions from referencing $\mathbf{s}_{pre}$, but the universal quantification works too.

unsharded transitions are described statefully, so an **add** followed by a **remove** is different from a **remove** followed by an **add**. On the other hand, in the sharded interpretation, the **remove** corresponds to an input token and the **add** corresponds to an output token, which most sensibly corresponds to putting the **remove** before the **add**. Putting them in the other order is strictly more permissive, thus not unsound; however, VerusSync disallows it anyway.

- The Init operations move around bulk collections of tokens. For example, for a storage_map field of type **map** k_i v_i, we input multiple tokens of type v_i. Likewise, for a map field, we *output* multiple tokens of type Tok_i. In practice, these bulk token operations are done with Verus's ghost collection type, `Map<K, TokenType>`.[2]

5.5 Soundness

The main theorem for the chapter is:

Theorem 1 *If a VerusSync Core System is well-formed in its unsharded interpretation, then the token types and operations of the sharded interpretation are implementable via Verus's Storage Protocol interface.*

In this section, we will sketch a proof. To build up to it incrementally, we start with an easier theorem:

Theorem 2 *If a VerusSync Core System is well-formed in its unsharded interpretation, and none of its fields use a storage strategy, then the token types and operations of the sharded interpretation are implementable via Verus's Resource Algebra interface.*

Note that VerusSync is part of the TCB of Verus; these are on-paper theorems, not mechanized.

5.5.1 Proof of Theorem 2 — Resource Algebras

Constructing a resource algebra Our first step to proving Theorem 2 is to construct an RA based on the fields and sharding strategies of the shardable state.

We first define several monoids which represent individual fields, starting by recalling $\textsc{Excl}(X)$ from Example 1:

$$\textsc{Excl}(X) \ni \epsilon \mid \mathsf{ex}(x) \mid \lightning \qquad \begin{array}{l}\text{with } \forall x, y.\ \mathsf{ex}(x) \cdot \mathsf{ex}(y) = \lightning \\ \text{and } \forall a,\ a \cdot \epsilon = a \text{ and } a \cdot \lightning = \lightning \\ \text{and } \forall x.\ \mathcal{V}(x) \Leftrightarrow (x \neq \lightning), \text{ all other elements valid}\end{array}$$

Next we define $\textsc{Agree}(X)$, which looks similar at first, but it allows duplicating its main proposition, and does not allow changing its value:

$$\textsc{Agree}(X) \ni \epsilon \mid \mathsf{ag}(x) \mid \lightning \qquad \begin{array}{l}\text{with } \forall x, y.\ \mathsf{ag}(x) \cdot \mathsf{ag}(y) = (\textbf{if } x = y \textbf{ then } \mathsf{ag}(x) \textbf{ else } \lightning) \\ \text{and } \forall a,\ a \cdot \epsilon = a \text{ and } a \cdot \lightning = \lightning \\ \text{and } \forall x.\ \mathcal{V}(x) \Leftrightarrow (x \neq \lightning), \text{ all other elements valid}\end{array}$$

[2]The actual VerusSync also supports bulk collection operations like this on the normal transitions as well.

$$
\begin{array}{rcl}
\textbf{require } e & \rightsquigarrow & \{\textbf{precondition}(e)\} \\
\textbf{assert } e & \rightsquigarrow & \{\textbf{postcondition}(e)\} \\
\textbf{init } f_i \leftarrow (e : \tau_i) \text{ for } \sigma_i \in \{\mathsf{variable}, \mathsf{constant}\} & \rightsquigarrow & \{\textbf{output}(\alpha : \mathsf{Tok}_i), \textbf{postcondition}(\alpha = e)\} \\
\textbf{init } f_i \leftarrow (e : \tau_i) \text{ for } \sigma_i \in \{\mathsf{map}, \mathsf{persistent\_map}\} & \rightsquigarrow & \{\textbf{output}(\alpha : \mathsf{Tok}_i), \textbf{postcondition}(\alpha = (e_1, e_2)) \mid (e_1, e_2) \in e\} \\
\textbf{init } f_i \leftarrow (e : \tau_i) \text{ for } \sigma_i = \mathsf{storage\_map} & \rightsquigarrow & \{\textbf{input}(\alpha : v_i), \textbf{precondition}(\alpha = e_2) \mid (e_1, e_2) \in e\} \\
\textbf{update } f_i \textbf{ from } (e : \tau_i) \textbf{ to } \text{for } \sigma_i = \mathsf{variable} & \rightsquigarrow & \{\textbf{input}(\alpha : \mathsf{Tok}_i), \textbf{output}(\alpha' : \mathsf{Tok}_i), \textbf{postcondition}(\alpha' = e)\} \\
\textbf{add } f_i \texttt{ += } ((e_1 : k_i), (e_2 : v_i)) \text{ for } \sigma_i = \mathsf{map} & \rightsquigarrow & \{\textbf{output}(\alpha : \mathsf{Tok}_i), \textbf{postcondition}(\alpha = (e1, e2))\} \\
\textbf{remove } f_i \texttt{ -= } ((e_1 : k_i), (e_2 : v_i)) \text{ for } \sigma_i = \mathsf{map} & \rightsquigarrow & \{\textbf{input}(\alpha : \mathsf{Tok}_i), \textbf{precondition}(\alpha = (e1, e2))\} \\
\textbf{union } f_i \texttt{ ∪= } ((e_1 : k_i), (e_2 : v_i)) \text{ for } \sigma_i = \mathsf{persistent\_map} & \rightsquigarrow & \{\textbf{output}(\alpha : \mathsf{Tok}_i), \textbf{postcondition}(\alpha = (e1, e2))\} \\
\textbf{deposit } f_i \texttt{ += } ((e_1 : k_i), (e_2 : v_i)) \text{ for } \sigma_i = \mathsf{storage\_map} & \rightsquigarrow & \{\textbf{input}(\alpha : v_i), \textbf{precondition}(\alpha = e_2)\} \\
\textbf{withdraw } f_i \texttt{ -= } ((e_1 : k_i), (e_2 : v_i)) \text{ for } \sigma_i = \mathsf{storage\_map} & \rightsquigarrow & \{\textbf{output}(\alpha : v_i), \textbf{postcondition}(\alpha = e_2)\} \\
\textbf{be } f_i \texttt{ == } (e : \tau_i) \text{ for } \sigma_i \in \{\mathsf{constant}, \mathsf{variable}\} & \rightsquigarrow & \{\textbf{input}(\alpha : \&\mathsf{Tok}_i), \textbf{precondition}(\alpha = e)\} \\
\textbf{have } f_i \ni ((e_1 : k_i), (e_2 : v_i)) \text{ for } \sigma_i \in \{\mathsf{map}, \mathsf{persistent\_map}\} & \rightsquigarrow & \{\textbf{input}(\alpha : \&\mathsf{Tok}_i), \textbf{precondition}(\alpha = (e_1, e_2))\} \\
\textbf{guard } f_i \ni ((e_1 : k_i), (e_2 : v_i)) \text{ for } \sigma_i = \mathsf{storage\_map} & \rightsquigarrow & \{\textbf{output}(\alpha : \&v_i), \textbf{postcondition}(\alpha = e_2)\}
\end{array}
$$

Figure 5.6: **Key mapping STML statements to components of a token operation.** *The variable α denotes a fresh variable name.*

Finally for a monoid M, define $\textsc{Func}(X, M)$ to be the monoid of functions $f : X \to M$, with composition and validity defined pointwise, i.e., $(f \cdot g)(x) = f(x) \cdot g(x)$ and $\mathcal{V}(f) = \forall x.\, \mathcal{V}(f(x))$.

Now, for each field $f_i : \tau_i$ with strategy σ_i we define M_i:

$$M_i \triangleq \begin{cases} \textsc{Excl}(\tau_i) & \text{if } \sigma_i = \mathsf{variable} \\ \textsc{Agree}(\tau_i) & \text{if } \sigma_i = \mathsf{constant} \\ \textsc{Func}(k_i, \textsc{Excl}(v_i)) & \text{if } \sigma_i = \mathsf{map} \text{ and } \tau_i = \mathbf{map}\ k_i\ v_i \\ \textsc{Func}(k_i, \textsc{Agree}(v_i)) & \text{if } \sigma_i = \mathsf{persistent\_map} \text{ and } \tau_i = \mathbf{map}\ k_i\ v_i \end{cases}$$

Finally, define M as the product monoid $M_1 \times \cdots \times M_n$. We are going to define the validity $\mathcal{V} : M \to \mathit{Prop}$ in a more restrictive way than usual.

For $m_i \in M_i$, define:

$$\mathsf{Complete}_i(m_i) \triangleq \begin{cases} \mathcal{V}(m_i) \wedge (m_i \neq \epsilon) & \text{if } \sigma_i \in \{\mathsf{constant}, \mathsf{variable}\} \\ \mathcal{V}(m_i) & \text{otherwise} \end{cases}$$

And for $m \in M$:

$$\mathsf{Complete}(m) \triangleq \mathsf{Complete}_1(m_1) \wedge \cdots \wedge \mathsf{Complete}_n(m_n)$$

In other words, $\mathsf{Complete}(m)$ essentially says, "each tuple entry of m is valid in its respective monoid, and also, for every constant and variable entry, that field has a value.

Now, we can construct a map from M restricted to the domain of complete values to the *unsharded state* S:

$$\mathsf{Unsh}_i : \{m_i : M_i \mid \mathsf{Complete}_i(m_i)\} \to \tau_i$$

$$\mathsf{Unsh}_i(m_i) \triangleq \begin{cases} x & \text{if } \sigma_i = \mathsf{variable} \text{ and } m_i = \mathsf{ex}(x) \\ x & \text{if } \sigma_i = \mathsf{constant} \text{ and } m_i = \mathsf{ag}(x) \\ \{k \mapsto x \mid m[k] = \mathsf{ex}(x)\} & \text{if } \sigma_i = \mathsf{map} \\ \{k \mapsto x \mid m[k] = \mathsf{ag}(x)\} & \text{if } \sigma_i = \mathsf{persistent\_map} \end{cases}$$

$$\mathsf{Unsh} : \{m : M \mid \mathsf{Complete}(m)\} \to S$$
$$\mathsf{Unsh}((m_1, \ldots, m_n)) = (\mathsf{Unsh}_1(m_1), \ldots, \mathsf{Unsh}_n(m_n))$$

(Note that one of the cases is always applicable if $\mathsf{Complete}_i(m_i)$.)

Recall (§5.3) that the unsharded interpretation is well-formed if we can find an invariant $\mathit{Inv} : S \to \mathit{Prop}$ satisfying the inductiveness criteria.

Finally, we state validity for M:

$$\mathcal{V}(m) \triangleq \exists d.\, \mathsf{Complete}(m \cdot d) \wedge \mathit{Inv}(\mathsf{Unsh}(m \cdot d))$$

Essentially, we are defining a set of "valid global states"—those which are complete and which satisfy the user-provided invariant—and then taking the $\preccurlyeq$-closure.

Interpretations of the tokens Now that we have finally defined a resource algebra, we can instantiate the ghost `Resource<P>` (Figure 4.4). We now need to specify interpretations for the token types. For each token T, we define a function $[\![T]\!] : \lfloor T \rfloor \to M$. Specifically:

$$[\![\mathsf{Instance}]\!](()) \triangleq (\epsilon, \ldots, \epsilon)$$

$$[\![\mathsf{Tok}_i]\!](x) \triangleq \begin{cases} \mathsf{Just}_i(\mathsf{ex}(x)) & \text{if } \sigma_i = \text{variable} \\ \mathsf{Just}_i(\mathsf{ag}(x)) & \text{if } \sigma_i = \text{constant} \\ \mathsf{Just}_i(\mathsf{FuncOne}(k, \mathsf{ex}(v))) & \text{if } \sigma_i = \text{map and } x = (k, v) \\ \mathsf{Just}_i(\mathsf{FuncOne}(k, \mathsf{ag}(v))) & \text{if } \sigma_i = \text{persistent\_map and } x = (k, v) \end{cases}$$

where $\mathsf{Just}_i(x)$ is the tuple with x at the i^{th} element, ϵ elsewhere, and $\mathsf{FuncOne}(k, x)$ is the function that maps k to x and everything else to ϵ.

Thus, we can now say that any token of type T with value x can be represented by a `Resource<P>` with value $[\![\mathsf{Tok}_i]\!](x)$. The only thing left to do is reason that all the operations can be implemented.

Implementing the operations At a high level:

- For any operation, all the owned input tokens are joined together with `join`. The tokens that are passed in by shared reference are joined together with `join_shared`.
- Any postcondition that results from an STML **assert** statement (whether in a Trans operation or a Property operation) follows from `validate_with_shared`.
- Use `update_nondeterministic_with_shared` to construct the output tokens for any Trans operation.
- Use `alloc` to construct the output tokens for any Init operation.
- The duplicable tokens can be duplicated by `update_with_shared`, using the fact that $[\![T]\!](x) \cdot [\![T]\!](x) = [\![T]\!](x)$.

Let us tackle the Trans operations in detail, as this is the most interesting aspect.

Transitions For simplicity, let's start with the deterministic case, i.e., where s_{pre} is not referenced. Recall our definition for validity:

$$\mathcal{V}(m) \triangleq \exists d.\ \mathsf{Complete}(m \cdot d) \wedge \mathit{Inv}(\mathsf{Unsh}(m \cdot d))$$

We can now expand the definition for a frame-preserving update:

$$\begin{aligned} a \rightsquigarrow b = \forall c.\ &(\exists d.\ \mathsf{Complete}(c \cdot a \cdot d) \wedge \mathit{Inv}(\mathsf{Unsh}(c \cdot a \cdot d))) \\ &\Rightarrow (\exists d.\ \mathsf{Complete}(c \cdot b \cdot d) \wedge \mathit{Inv}(\mathsf{Unsh}(c \cdot b \cdot d))) \end{aligned}$$

By collapsing c and d, we get a slightly weaker condition that implies $a \rightsquigarrow b$:

$$\forall c.\ (\mathsf{Complete}(a \cdot c) \wedge \mathit{Inv}(\mathsf{Unsh}(a \cdot c))) \Rightarrow (\mathsf{Complete}(b \cdot c) \wedge \mathit{Inv}(\mathsf{Unsh}(b \cdot c)))$$

In order to apply `update_with_shared`, we need to show that $a \cdot x \rightsquigarrow b \cdot x$ where a is the composition of the owned input tokens, x is the (overlapping) conjunction of the shared reference input tokens, and b is the composition of the output tokens.

Thus, given $\mathsf{Complete}(a \cdot x \cdot c) \wedge \mathit{Inv}(\mathsf{Unsh}(a \cdot x \cdot c))$ and all the relevant preconditions on a and x, we have two tasks: We need to show $\mathsf{Complete}(b \cdot x \cdot c)$ and we need to show $\mathit{Inv}(\mathsf{Unsh}(b \cdot x \cdot c))$.

- $\mathsf{Complete}(b \cdot x \cdot c)$ follows from the structure of the token generation and from the *safety conditions* that are produced in the STML $\rightarrow$ SimpleRML translation (Figure 5.4).
  - For example, for a variable token, we always either **(i)** take it as a shared reference, or **(ii)** take an owned token as input and return an owned token. Thus the $\mathsf{ex}(x)$ element either appears in both a and b, or it appears in x.
  - For a map or persistent_map field, the challenge is to show, when creating a new token, that it does not result in some conflict $\lightning$ appearing in the output. For example, suppose the user writes $\mathbf{add}\ f_i\ \texttt{+=}\ (e_1, e_2)$ for some map field. We need to show that e_1 does not conflict with an existing key; this follows from the safety condition $\mathbf{assert}\ e_1 \notin f_i$ in the STML $\rightarrow$ SimpleRML translation. More precisely, the completeness of the post-state, $\mathsf{Complete}(b \cdot x \cdot c)$, follows from the fact that, by the definition of well-formedness, the invariant implies all the safety conditions.
- $\mathit{Inv}(\mathsf{Unsh}(b \cdot x \cdot c))$ follows from the inductiveness criterion of *Inv*.

Extending to nondeterministic updates Now, let's briefly look at the nondeterministic case. Suppose the operation definition references $\mathbf{s}_{pre}$. Since we do not know $\mathbf{s}_{pre}$ a priori, we need to perform a nondeterministic update. The postcondition of the operation has the form $\forall \mathbf{s}_{pre}.\ \mathit{Post}(\mathbf{s}_{pre})$. The target combined value of the output tokens can then be written as a function of $\mathbf{s}_{pre}$, say, $\beta(\mathbf{s}_{pre})$.

Recall the definition of a nondeterministic update:

$$\begin{aligned} a \rightsquigarrow B = \forall c.\ &(\exists d.\ \mathsf{Complete}(c \cdot a \cdot d) \wedge \mathit{Inv}(\mathsf{Unsh}(c \cdot a \cdot d))) \\ &\Rightarrow \exists b.\ b \in B \wedge (\exists d.\ \mathsf{Complete}(c \cdot b \cdot d) \wedge \mathit{Inv}(\mathsf{Unsh}(c \cdot b \cdot d))) \end{aligned}$$

Let $B = \{b \mid \exists \mathbf{s}_{pre}.\ b = \beta(\mathbf{s}_{pre})\}$. Then we have,

$$\begin{aligned} a \rightsquigarrow B = \forall c.\ &(\exists d.\ \mathsf{Complete}(c \cdot a \cdot d) \wedge \mathit{Inv}(\mathsf{Unsh}(c \cdot a \cdot d))) \\ &\Rightarrow \exists \mathbf{s}_{pre}.\ (\exists d.\ \mathsf{Complete}(c \cdot \beta(\mathbf{s}_{pre}) \cdot d) \wedge \mathit{Inv}(\mathsf{Unsh}(c \cdot \beta(\mathbf{s}_{pre}) \cdot d))) \end{aligned}$$

It is enough to show this for a specific value of $\mathbf{s}_{pre}$, in terms of c. By introducing $\mathbf{s}_{pre}$ as $\mathsf{Unsh}(c \cdot a \cdot d)$, it is enough to show:

$$\begin{aligned} &\forall c.\ (\exists d.\ \mathsf{Complete}(c \cdot a \cdot d) \wedge \mathit{Inv}(\mathsf{Unsh}(c \cdot a \cdot d))) \\ &\quad \Rightarrow (\exists d.\ \mathsf{Complete}(c \cdot \beta(\mathsf{Unsh}(c \cdot a \cdot d)) \cdot d) \wedge \mathit{Inv}(\mathsf{Unsh}(c \cdot \beta(\mathsf{Unsh}(c \cdot a \cdot d)) \cdot d))) \end{aligned}$$

Again, collapse $c \cdot d$ to a single variable. It is enough to show:

$$\begin{aligned} &\forall c.\ (\mathsf{Complete}(c \cdot a) \wedge \mathit{Inv}(\mathsf{Unsh}(c \cdot a))) \\ &\quad \Rightarrow (\mathsf{Complete}(c \cdot \beta(\mathsf{Unsh}(c \cdot a))) \wedge \mathit{Inv}(\mathsf{Unsh}(c \cdot \beta(\mathsf{Unsh}(c \cdot a))))) \end{aligned}$$

From here, the same logic as above applies, using the safety conditions and inductiveness criteria to to check $\mathsf{Complete}(c \cdot \beta(\mathsf{Unsh}(c \cdot a)))$ and $\mathit{Inv}(\mathsf{Unsh}(c \cdot \beta(\mathsf{Unsh}(c \cdot a))))$.

5.5.2 Proof of Theorem 1 — Extending to Storage Protocols

Now, let's discuss how to handle fields with the strategy storage_map. Without loss of generality, we'll assume we have just one storage_map field. We can extend this to handle multiple (e.g., we can merge all the storage fields into one, using an enum for the map keys). Let this field be called f_{store}, with type $\tau_{\text{store}} = \mathbf{map}\ k_{\text{store}}\ v_{\text{store}}$. Let the other fields be $f_1, \ldots, f_n$.

In accordance with §5.4, we are not supposed to generate any "tokens" for a storage_map field. Thus, we define M to be $M_1 \times \cdots \times M_n$. Note that there is no M_{store}.

We can define the $\mathsf{Complete}_i$ and Complete predicates the same way as before. However, we cannot define Unsh as before since M does not have the value of the storage field. Thus we append a second argument to Unsh:

$$\mathsf{Unsh} : \{m : M \mid \mathsf{Complete}(m)\} \times \tau_{\text{store}} \to S$$
$$\mathsf{Unsh}((m_1, \ldots, m_n), s) = (\mathsf{Unsh}_1(m_1), \ldots, \mathsf{Unsh}_n(m_n), s)$$

Recall that for a storage protocol, we need to define a relation between the protocol monoid and the storage monoid which is meant to implicitly encode all the validity conditions.

$$\mathcal{R} : P \times \mathbf{map}\ k_{\text{store}}\ v_{\text{store}} \to \mathit{Prop}$$
$$\mathcal{R}(m, s) \triangleq \mathsf{Complete}(m) \wedge \mathit{Inv}(\mathsf{Unsh}(m, s))$$

Instantiating the trait, we get the `StorageResource` tokens.

Now, in general, a transition might withdraw or deposit the ghost objects of type v_i (using the STML **withdraw** or **deposit**) statements. Thus, the transition can be encoded into `exchange_nondeterministic_with_shared`. We can analyze the exchange transition $\dashrightarrow$ (Figure 4.9a) similarly to the above.

Of course, the unique feature of interest is the guarding. Recall that a statement like

$$\mathbf{guard}\ f_i \ni ((e_1 : k_i), (e_2 : v_i))$$

can only appear in a Property operation, so all our inputs are shared references to ghost tokens. We can conjoin the inputs with `join_shared` as usual and apply `guard`. Furthermore, a well-formedness condition on **guard** prevents e_1 or e_2 from depending on $\mathbf{s}_{pre}$, so both expressions are deterministic in the parameters of the transition.

Recall the definition of $\twoheadrightarrow$ (Figure 4.9a), the condition for deducing a guard:

$$p \twoheadrightarrow s \triangleq \forall q, t.\ \mathcal{R}(p \cdot q, t) \Rightarrow s \preceq t$$

Ultimately, we need to show $g \twoheadrightarrow [e_1 \mapsto e_2]$ where g is the conjunction of all the input tokens. Note that this is well-formed since by the requirements in Figure 5.2, $\mathbf{s}_{pre}$ is not free in e_1 or e_2. Thus we need:

$$\forall q, t.\ \mathcal{R}(g \cdot q, t) \Rightarrow [e_1 \mapsto e_2] \preceq t$$

Continuing to expand, we need:

$$\forall q, t.\ \mathsf{Complete}(g \cdot q) \wedge \mathit{Inv}(\mathsf{Unsh}(g \cdot q, t)) \Rightarrow e_1 \in t \wedge t[e_1] = e_2$$

Finally, this follows from the safety condition, **assert** $e_1 \in f_i \wedge f_i[e_1] = e_2$ in the STML $\rightarrow$ SimpleRML translation (Figure 5.4).

5.6 Why VerusSync over resource algebras?

The reader might still be wondering what we gain here over a resource algebra. In comparison to the resource algebra concept, VerusSync admittedly seems kind of weird. The resource algebra is so powerful as a foundation for CSL in part because it is elegant and canonical, while VerusSync is neither elegant nor canonical. In fact, VerusSync might be the least canonical, most *ad hoc* thing I've ever invented.

I created VerusSync by taking a bunch of features that seemed useful from my experience in IronSync and threw them into a big pot of soup. I chose features that intuitively could be formalized in an RA, and I implemented syntax for these features so that they could be invoked in only a couple of lines. As a foundation for concurrent separation logic, VerusSync would be terrible, but it's not a foundation for concurrent separation logic. VerusSync is a swiss army knife for concurrent program verification.

It's easy to compare VerusSync to a resource algebra, but this slightly misses the point, I think. As stated at the beginning of the chapter, VerusSync is about more than just *composition*. It's about the *global state* of a system; it's about organizing the *invariants*, *transitions*, and (when storage is involved) *guarding* properties. And one can't forget: it's about bashing through the proof obligations with an automated theorem prover.

5.7 Recap

We formalized VerusSync by defining a simple language for a state transition system with two interpretations. The first interpretation is similar to prior work on modeling systems as transition systems, which is both intuitive for the developer and which enables us to take advantage of techniques for generating efficient verification conditions. The other interpretation allows us to create a useful ghost state API.

Chapter 6

Type System, Primitive Specifications, and Soundness

In this chapter, we will introduce a formal language and type system, λ_{Verus}, that includes most of the Verus primitives types along with formal specifications for their primitive operations. We will also sketch how to prove these type-specifications sound using Iris.

This is the most technically involved chapter of the thesis. This chapter continues to use Iris in advanced ways, and it builds substantially on RustBelt's λ_{Rust} [33], prior work for reasoning about soundness in Rust's type system. Due to the complexity of each, this chapter will be less self-contained than the others, as I will refer extensively to the prior work.

6.1 λ_{Verus} scope

With λ_{Verus} we will be able to handle:

- Heap pointers and ghost permissions
- Interior mutability and ghost permissions
- The Verus Monoidal Ghost Interface, including the RA interface and the storage protocol interface
- `LocalInvariant`
- `AtomicInvariant` and atomic instructions (but see the caveats below)
- Shared references and lifetimes, including shared references to ghost state

We will not attempt to capture any of the following features:

- Mutable references
- Prophecy variables
- Verus's specification language or proof language (outside of `Tracked` types)
- Control flow in ghost code, or the ability to "make decisions" in ghost code
- The termination of ghost code

However, it should be noted that the Verus paper [42] already addresses the formalization of

Verus's mode system, including the specification language, and in particular, the termination of spec code. It does so by a translation into the Calculus of Inductive Constructions (CoIC), which we take to be the "meta-logic" for this chapter. Thus, we will treat specifications via a shallow embedding in the meta-logic.

On ghost code termination Of all our caveats, the one that needs the most explanation is the last bullet point, the termination of ghost code. This has a few implications:

- Ordinarily, we would hope to make some claim like: "If a λ_{Verus} program p is well-typed, and p' is its *erased* program, i.e., p' is p with all ghost code removed, then p' executes with no undefined behavior." This claim, the *erasure* claim, would be the gold standard for any erasure-based system. Unfortunately, the erasure theorem does not hold unless we have a guarantee that the ghost code between any two executable "physical" instructions is terminating. Otherwise, we could have p execute an infinite loop of ghost code, followed by an instruction that exhibits undefined behavior. This would be well-typed and well-specced in λ_{Verus}, but p' would exhibit undefined behavior.
- A more complex issue concerns *atomic invariants*. Recall that in Verus, when an atomic invariant is open, the program is allowed to execute one atomic instruction in addition to *arbitrary terminating ghost code*. If not for the termination requirement, one could easily establish a contradiction as follows:
  - Established a shared memory location with an atomic invariant that the value in the location is either 0 or 1.
  - Open an atomic invariant, access the location, and write the value 2.
  - Go into an infinite loop of ghost code.
  - Prove false in order to re-establish the invariant, and close the block.
  - On *another thread*, read from the memory location, and wrongly conclude that the value is not 2.

In order to avoid dealing with all of these problems, we will give λ_{Verus} some slightly odd semantics. Specifically, we will not bother distinguishing between "ghost code" and "physical code" at all. Some instructions will operate on ghost *values*, but these will still considered ordinary physical instructions. Thus an infinite loop of "ghost code" is just an ordinary infinite loop. To resolve the second problem, we will say that opening an atomic invariant takes a "global lock" which prevents any other threads from accessing the heap. Thus, if any atomic invariant block has an infinite loop in it, then the semantics of the language dictate that this will stall all threads.

Now, if a developer wants to make an argument that all ghost code is terminating, and that all atomic blocks have at most one executable instruction, and that therefore it is permissible to erase all ghost code and erase the global lock (i.e., to perform the standard compilation strategy) then they can do so; however, this is an argument that they will need to make externally to λ_{Verus}'s type system.

Other caveats It should be noted that nothing in this section has been mechanized in Coq, nor have I exhaustively checked every last rule by hand. This is, admittedly, somewhat risky for an Iris proof, as Iris is a fairly subtle logic with difficult intuitions, and LaTeX is not renowned for its dependent type-checking capabilities. However, I believe this suffices for my primary aim of the section: to elucidate both the essence of our methodology and the connections between Leaf and Verus.

6.2 Method overview and background

One of the foundational works in the Rust verification is RustBelt [33], which defines a formal language, λ_{Rust}. Though simplified, it captures most of what is interesting about Rust's lifetimes and references. RustBelt uses a *logical relation* to prove λ_{Rust}'s type safety, RustBelt's work is notable for its extensibility and its ability to handle the encapsulation of unsafe Rust code.

As approaches to Rust verification based on first-order logic specifications gained traction, RustHornBelt [56] entered the scene and introduced *type-specs* to λ_{Rust}—typing rules equipped with *specifications* along with a proof of soundness of these specifications.

Building off of λ_{Rust} is appealing for a few reasons. For one, its scoped feature set is close to what we want to handle. Like Verus, it handles SC+NA memory semantics. More importantly, the formal proofs of RustBelt and RustHornBelt are executed in Iris, which makes it a great choice for handling all of the Iris-inspired concepts that Verus uses. And finally, RustHornBelt's type-spec system is well-suited to formalizing the specifications of all the Verus primitives—the memory permissions, invariant types, and the Verus Monoidal Ghost Interface.

With this in mind, our high level plan is as follows. We will first define λ_{Verus} as an extension of λ_{Rust} in order to incorporate Verus's primitives. Then, we will apply RustHornBelt's methodology[1]—its approach to incorporating specifications into typing rules, its semantic interpretations of these typing rules, and its logical relations theorem—albeit with some tweaks to integrate Leaf.

6.3 λ_{Verus} syntax, semantics, and specifications

The syntax and types for λ_{Verus} is shown in Figure 6.1. Most of it is verbatim from λ_{Rust}; the important additions we have made are highlighted in blue.

As you can see, we have also struck out **mut**, since we are not handling it here. We will have to work around this, since many Verus primitives of interest (e.g., `PCell` in Figure 3.2) use `&mut` references. To handle such cases, we will replace `&mut T` references with a separate in-parameter and out-parameter, each of type `T`.

[1]It should be noted that one of RustHornBelt's most novel and well-known contribution has to do with the use of prophecy variables to handle mutable references, so I should emphasize again that we are not going to be handling mutable references in this chapter. The main element we use from RustHornBelt is the type-spec system.

6.3.1 The λ_{Verus} language and operational semantics

Elements inherited from λ_{Rust} In λ_{Rust} and λ_{Verus}, the "value" types are all very simple: they include primitive integers, bools, pointers, and functions. More complex objects like products and sums only exist in the memory heap. The memory heap is a collection of "memory cells" indexed by integer addresses, and any object occupies a contiguous range of memory. For example, a pair of two integers would occupy two consecutive memory cells. Since all data is kept in memory like this, even what we would usually think of as "stack variables," we can always take pointers and references to the data.

The instructions operate on *paths* $p.n$ which just add n to the pointer. For example, if p points to a pair of integers, we might do $p.0$ to get a pointer to the first integer or $p.1$ to get a pointer to the second. Sums are represented as tagged unions with the tag in the first cell ($p.0$) and the inner contents starting immediately after ($p.1$).

If p_1 and p_2 are both pointers, and you want to copy an object of size n from p_2 to p_1, you would write $p_1 :=_n {}^*p_2$. This is used for ordinary "moves" and "copies" in Rust. Figure 6.2 shows how copying objects around is done via non-atomic "memcpy."

For the full details of the operational semantics, I refer to the RustBelt technical appendix [34]. This appendix fully defines the semantics of the core lambda calculus, including its memory model, and it spells out the full translation of λ_{Verus} into this core calculus, including the continuation-passing strategy used for control flow.

There are a few details we need to cover on the memory model. The core lambda calculus provides two *memory orderings*, sc for sequentially-consistent atomic memory ordering, and na for "non-atomic" memory. The name is quite literal: a non-atomic read or write takes multiple atomic steps in the operational semantics. It is defined in a specific way so as to detect *data races*—conflicting reads or writes to the same location. A data race is defined to be a "stuck" state—thus, proving safety of an execution implies freedom from data races. Most instructions use the non-atomic memory ordering, as we can see, e.g., in the operational definition of $p_1 :=_n {}^*p_2$ in Figure 6.2.

λ_{Verus} new features Figure 6.2 also shows the semantics of λ_{Verus} primitive operations, which we treat as instructions. For the most part, the semantics of these operations are straightforward. On one hand, we have a handful of ghost operations (RA operations, storage protocol operations, invariants) that are no-ops (although they *do* formally "take a step").

The pointer and cell operations actually manipulate memory; also, since cells are only meaningful in the type system the cell operations actually have the same semantic meaning as the pointer operations. Also observe that **PPtr_borrow** and **PCell_borrow** (corresponding to `PPtr::borrow` and `PCell::borrow` respectively) just pass a pointer through unchanged. This operation is only interesting in the type system, where a `PPtr` or a `&PCell<T>` becomes a `&T`.

The last thing to point out is that **Atomic_Open** and **Atomic_Close** differ from the rest of the ghost operations in that they acquire and release the "global atomic lock," for the reasons described earlier. We treat this lock reentrantly—i.e., one thread can take the lock multiple times. This lets the user open multiple atomic invariants without deadlocking.

Contexts of the λ_{Verus} type system The top-level context, Γ, gives the sort for each variable, indicating it as either a value, a lifetime variable, or a type variable. A *lifetime*, κ is either a lifetime variable α (written `'a` in Rust) or the **static** lifetime, i.e., the lifetime that lasts for the duration of the program.

E and **L** are the *external lifetime context*, and the *local lifetime context*, respectively. The external lifetime context contains lifetime constraints from the signature of the current function, where $\kappa \sqsubseteq_e \kappa'$, indicates, "$\kappa'$ outlives κ" or "as long as κ is alive, κ' is alive as well." Meanwhile, the local lifetime context contains lifetime constraints from checking the body of a function. In a local lifetime inclusion, $\kappa \sqsubseteq_e \overline{\kappa}$, the symbol $\overline{\kappa}$ is a vector of lifetime variables, and the inclusion indicates that κ is alive *if* all lifetimes in $\overline{\kappa}$ are alive. The main difference between the two is that the external lifetime context is fixed for the duration of a function, i.e., a constraint $\kappa \sqsubseteq_e \kappa'$ can be assumed to hold true *always* after the current program point, whereas a local lifetime inclusion $\kappa \sqsubseteq_l \overline{\kappa}$ is substructural.

RustBelt's *modifier* μ usually indicates whether a borrow is shared or mutable. Again, we are not covering mutable borrows, so in our case, it is somewhat vestigial.

In the *typing context*, **T**, there are two kinds of type annotations. First, the "normal" judgment $p \lhd \tau$ indicates that p has type τ Meanwhile, $p \lhd^{\dagger\kappa} \tau$ means that p has type τ but that p is currently *borrowed from* via some borrow associated with lifetime κ, and that p is inaccessible until κ has expired.

Next, and novel to λ_{Verus}, we have the *invariant context*, which tracks then open invariants in order to ensure that every **Atomic_Close** corresponds to a **Atomic_Open** and every **Local_Close** to a **Local_Open**.

Finally, we have the *continuation context* **K** with continuation judgments $k \lhd \mathbf{cont}(\mathbf{L}; \mathbf{I}; \overline{x}.\ \mathbf{T})$ which says that k can be called given the appropriate local lifetime, typing, and invariant contexts.

6.3.2 Types of λ_{Verus}

λ_{Verus} types include:

- Standard type constructs such as primitive types **int** and **bool**, sum types $\Sigma\overline{\tau}$, product types $\Pi\overline{\tau}$, and recursive types $\mu T.\ \tau$.
- $\lightning_n$, representing uninitialized, arbitrary memory spanning n memory words.
- The function type $\forall\overline{\alpha}.\ \mathbf{fn}(\mathsf{F} : \mathbf{E}; \overline{\tau}) \to \tau$. The function type is polymorphic over lifetime variables. (λ_{Verus}, like λ_{Rust}, does not have explicit polymorphic *type* variables; instead we just quantify over types in the meta-logic.)
- $\mathbf{own}_n\ \tau$ is the type of a pointer that points to n memory words that together contain a value of type τ. This type roughly corresponds to Rust's `Box<T>` type, though it is also used for stack variables. For example, if a Rust program has a physical stack variable X, then the corresponding λ_{Verus} program will have $p \lhd \mathbf{own}_n\ X$ in the typing context. Putting stack variables on the heap allows us to take references to them.
- The reference type $\&^{\kappa}_{\text{shr}}\ \tau$—a shared reference to a type τ with lifetime κ.
- Types specific to λ_{Verus}:
  - **Tracked** τ is kind of like the "ghost" version of **own** and corresponds to Verus's

`Tracked`. We can wrap any type in `Tracked` to make the ghost version of it.

To be *extremely pedantic*, we also need these types to be "on the heap, but taking up 0 bytes of it." This reduces the need for lots of special casing by letting us treat **Tracked** τ like any other type. That said, adding $\mathbf{own}_0$ everywhere will get old fast, so we use an abbreviation:

$$\mathbf{trk}\ \tau \triangleq \mathbf{own}_0\ \mathbf{Tracked}\ \tau$$

Thus, **trk** τ is how we will represent "ownership of ghost values."

How do we represent shared references of ghost values? We could represent it as $\&^{\kappa}_{\text{shr}}\mathbf{Tracked}\ \tau$, obviously, but this causes us to carry around a meaningless pointer. We prefer to have the the **Tracked** on the outside. Again, we create a shorthand:

$$\&^{\kappa}_{\text{trk}}\tau \triangleq \mathbf{trk}\ \&^{\kappa}_{\text{shr}}\tau$$

- **PPtr**, $\mathbf{PointsTo}_n\ \tau$, and $\mathbf{Dealloc}_n$ correspond to `PPtr`, `PointsTo<T>`, and `Dealloc<T>`, respectively. I won't be handling `PointsToRaw`, Verus's type for variable-sized memory. Like in $\mathbf{own}_n\ \tau$, we track the number of memory words in the allocation n. Also note that **PPtr** takes no type parameter; the pointer itself does not need information about the type of the allocation, which is all in the $\mathbf{PointsTo}_n\ \tau$.
- $\mathbf{PCell}_n$ and $\mathbf{Cell::PointsTo}_n\ \tau$, the analogue of the above for cells. Note that $\mathbf{PCell}_n$ does need the type size n, since this determines the size of the memory representation, but again, it does not need the type of the contained value.
- $\mathbf{Resource}(RA)$ and $\mathbf{StorageResource}(\tau, SP)$, parameterized by resource algebra and storage protocols respectively.
- $\mathbf{AtomicInvariant}(C, \tau, I)$ and $\mathbf{LocalInvariant}(C, \tau, I)$. The three arguments here correspond to the three type parameters of the Verus invariant types. C is an arbitrary sort in the meta-logic and I is a predicate $C \times \lfloor\tau\rfloor \to \mathit{Prop}$. (Here, $\lfloor\cdot\rfloor$ is the encoding's *representation* of the type τ, which we will introduce more formally soon.)

6.3.3 λ_{Verus} type-spec judgments

RustHornBelt introduces a *type-spec* judgment to RustBelt, so that instructions can be typed together with *specifications* that allow users to reason about the correctness of the program in first-order logic.

To explain the type specs, we first need to explain how different types are represented in specifications. Formally, for each type τ, we define a *representation sort* $\lfloor\tau\rfloor$. The sort $\lfloor\tau\rfloor$ is the sort of the variables in the encoding used to represent values of type τ. The definitions of $\lfloor\cdot\rfloor$ are given in Figure 6.3. We see that an **int** for example is just represented as an integer in $\mathbb{Z}$. Rust's standard pointer types, $\mathbf{own}_n$ and $\&^{\kappa}_{\text{shr}}$, are represented the same way as their pointee. We will dive further into the Verus-specific types later.

Now that we have representations for the types, we can talk about the specifications themselves. The type-spec judgment for an instruction is written:

$$\mathbf{E}; \mathbf{L} \mid \mathbf{I}; \mathbf{T} \vdash I \dashv r.\, \mathbf{I}'; \mathbf{T}' \rightsquigarrow \Phi$$

λ_{Verus} syntax

$$
\begin{aligned}
\mathit{Path} \ni p &::= x \mid p.n \\
\mathit{Val} \ni v &::= \mathsf{False} \mid \mathsf{True} \mid z \mid \ell \mid \mathbf{funrec}\ f(\overline{x})\ \mathbf{ret}\ k := F \\
\mathit{Instr} \ni I &::= \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 \le p_2 \mid p_1 = p_2 \mid \mathbf{new}(n) \mid \mathbf{delete}(n, p) \\
&\quad \mid {}^*p \mid p_1 := p_2 \mid p_1 :=_n {}^*p_2 \\
&\quad \mid p :\overset{\text{inj } i}{=\!=} () \mid p_1 :\overset{\text{inj } i}{=\!=} p_2 \mid p_1 :\overset{\text{inj } i}{=\!=}_n {}^*p_2 \mid \ldots \\
&\quad \mid (\text{Instructions for Verus primitive operations (See Figure 6.2)}) \\
\mathit{FuncBody} \ni F &::= \mid \mathbf{let}\ x = I\ \mathbf{in}\ F \mid \mathbf{letcont}\ k(\overline{x}) := F_1\ \mathbf{in}\ F_2 \mid \mathbf{newlft};\ F \mid \mathbf{endlft};\ F \\
&\quad \mid \mathbf{if}\ p\ \mathbf{then}\ F_1\ \mathbf{else}\ F_2 \mid \mathbf{case}\ {}^*p\ \mathbf{of}\ \overline{F} \mid \mathbf{jump}\ k(\overline{x}) \mid \mathbf{call}\ f(\overline{x})\ \mathbf{ret}\ k
\end{aligned}
$$

λ_{Verus} contexts and types

$$
\begin{aligned}
\mathit{Sort} \ni \sigma &::= \mathbf{val} \mid \mathbf{lft} \mid \mathbf{type} \\
\Gamma &::= \emptyset \mid \Gamma, X : \sigma \\
\mathit{Lft} \ni \kappa &::= \alpha \mid \mathbf{static} \\
\mathbf{E} &::= \emptyset \mid \mathbf{E}, \kappa \sqsubseteq_e \kappa' \\
\mathbf{L} &::= \emptyset \mid \mathbf{L}, \kappa \sqsubseteq_l \overline{\kappa} \\
\mathit{Mod} \ni \mu &::= \sout{\mathbf{mut}} \mid \mathbf{shr} \\
\mathit{GhostMod} \ni \eta &::= \mathbf{ghost} \mid \mathbf{phys} \\
\mathbf{T} &::= \emptyset \mid \mathbf{T}, p \lhd \tau \mid \mathbf{T}, p \lhd^{\dagger\kappa} \tau \\
\mathbf{I} &::= \emptyset \mid \mathbf{I}, \iota \lhd \mathbf{InAtomic}(C, \tau, I) \mid \mathbf{I}, \iota \lhd \mathbf{InLocal}(C, \tau, I) \\
\mathbf{K} &::= \emptyset \mid \mathbf{K}, k \lhd \mathbf{cont}(\mathbf{L}; \mathbf{I}; \overline{x}.\ \mathbf{T})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Type} \ni \tau &::= T \mid \mathbf{bool} \mid \mathbf{int} \mid \lightning_n \\
&\quad \mid \mathbf{own}_n\ \tau \mid \&^\kappa_\mu \tau \mid \Sigma\overline{\tau} \mid \Pi\overline{\tau} \mid \forall\overline{\alpha}.\ \mathbf{fn}(\varphi : \mathbf{E}; \overline{\tau}) \to \tau \mid \mu T.\ \tau \\
&\quad \mid \mathbf{Tracked}\ \tau \\
&\quad \mid \mathbf{PPtr} \mid \mathbf{PointsTo}_n\ \tau \mid \mathbf{Dealloc}_n \\
&\quad \mid \mathbf{PCell}_n \mid \mathbf{Cell::PointsTo}_n\ \tau \\
&\quad \mid \mathbf{Resource}(\mathit{RA}) \mid \mathbf{StorageResource}(\tau, \mathit{SP}) \\
&\quad \mid \mathbf{AtomicInvariant}(C, \tau, I) \mid \mathbf{LocalInvariant}(C, \tau, I)
\end{aligned}
$$

Figure 6.1: λ_{Verus} **syntax.** *Additions (as compared to RustBelt's λ_{Rust}) are highlighted in blue.*

λ_{Rust} copies and assignments

$$
\begin{array}{rcl}
\texttt{memcpy} & \triangleq & \texttt{rec}\ \mathit{memcpy}(\mathit{dst}, \mathit{len}, \mathit{src}) := \\
& & \quad \texttt{if}\ \mathit{len} \leq 0\ \texttt{then undefined else} \\
& & \quad \mathit{dst}.0 := \mathit{src}.0 \\
& & \quad \mathit{memcpy}(\mathit{dst}.1, \mathit{len} - 1, \mathit{src}.1)
\end{array}
$$

$$
\begin{array}{rcl}
{}^{*}e & \triangleq & {}^{*\mathsf{na}}e \\
e_1 := e_2 & \triangleq & e_1 :=_{\mathsf{na}} e_2 \\
e_1 :=_n {}^{*}e_2 & \triangleq & \texttt{memcpy}(e_1, n, e_2) \\
e :\overset{\text{inj } i}{=\!=} () & \triangleq & e.0 := i \\
e_1 :\overset{\text{inj } i}{=\!=} e_2 & \triangleq & e_1.0 := i \ ; \ e_1.1 := e_2 \\
e_1 :\overset{\text{inj } i}{=\!=}_n {}^{*}e_2 & \triangleq & e_1.0 := i \ ; \ e_1.1 :=_n {}^{*}e_2
\end{array}
$$

λ_{Verus} pointers and cells

PCell_new$_n$	$\triangleq$ **PPtr_new**$_n$	$\triangleq$ $\texttt{malloc}(n)$
PCell_destroy$_n\ e$	$\triangleq$ **PPtr_destroy**$_n\ e$	$\triangleq$ $\texttt{delete}(n, e)$
PCell_put$_n\ e_1\ e_2$	$\triangleq$ **PPtr_put**$_n\ e_1\ e_2$	$\triangleq$ $e_1 :=_n {}^{*}e_2\ ;\ \texttt{delete}(n, e_2)$
PCell_take$_n\ e$	$\triangleq$ **PPtr_take**$_n\ e$	$\triangleq$ $\texttt{malloc}(n) :=_n {}^{*}e$
PCell_borrow e	$\triangleq$ **PPtr_borrow** e	$\triangleq$ e

λ_{Verus} atomics

$$
\begin{array}{rcl}
\textbf{PPtr\_FAA}\ e\ i & \triangleq & \texttt{FAA}^{\texttt{sc}}(e, i) \\
\ldots & &
\end{array}
$$

λ_{Verus} ghost state

$$
\begin{array}{rcl}
\textbf{RA\_Alloc}, \textbf{RA\_Join}, \textbf{RA\_Split}, & & \\
\textbf{RA\_Unit}, \textbf{RA\_WeakenShared}, & & \\
\textbf{RA\_JoinShared}, \textbf{RA\_Validate}, \textbf{RA\_Update}, & & \\
\textbf{SP\_Guard}, \textbf{SP\_Exchange\_With\_Shared}, \ldots & \triangleq & \texttt{skip}
\end{array}
$$

λ_{Verus} invariants

Local_New	$\triangleq$ `skip`	**Atomic_New**	$\triangleq$ `skip`
Local_Destroy	$\triangleq$ `skip`	**Atomic_Destroy**	$\triangleq$ `skip`
Local_Open	$\triangleq$ `skip`	**Atomic_Open**	$\triangleq$ `acquire_reentrant_global_atomic_lock`
Local_Close	$\triangleq$ `skip`	**Atomic_Close**	$\triangleq$ `release_reentrant_global_atomic_lock`

Figure 6.2: **λ_{Verus} semantics of instructions.** *Instructions in λ_{Verus}/λ_{Rust} are defined in terms of a core lambda calculus with na and sc memory orderings.*

Context Interpretations

$$
\begin{aligned}
&\lfloor \mathbf{T} \rfloor && \triangleq \Pi_{t \in \mathbf{T}} \lfloor t \rfloor \\
&\lfloor p \lhd \tau \rfloor && \triangleq \lfloor \tau \rfloor \\
&\lfloor p \lhd^{\dagger \kappa} \tau \rfloor && \triangleq \lfloor \tau \rfloor \\
\\
&\lfloor \mathbf{I} \rfloor && \triangleq (\Pi_{i \in \mathbf{I}} \lfloor i \rfloor) \times \mathcal{P}(\mathit{Name}) \\
&\lfloor \iota \lhd \mathbf{InLocal}(C, \tau, I) \rfloor && \triangleq \mathit{Name} \times C \\
&\lfloor \iota \lhd \mathbf{InAtomic}(C, \tau, I) \rfloor && \triangleq \mathit{Name} \times C
\end{aligned}
$$

Type Interpretations

$$
\begin{aligned}
&\lfloor \mathbf{int} \rfloor && = \mathbb{Z} \\
&\lfloor \mathbf{bool} \rfloor && = \mathbb{B} \\
&\lfloor \lightning_n \rfloor && = \mathbf{unit} \\
&\lfloor \mathbf{own}_n\ \tau \rfloor && = \lfloor \tau \rfloor \\
&\lfloor \&^{\kappa}_{\mathrm{shr}}\ \tau \rfloor && = \lfloor \tau \rfloor \\
&\lfloor \Sigma \overline{\tau} \rfloor && = \Sigma \lfloor \tau \rfloor \\
&\lfloor \Pi \overline{\tau} \rfloor && = \Pi \lfloor \tau \rfloor \\
&\lfloor \mathbf{Tracked}\ \tau \rfloor && = \lfloor \tau \rfloor \\
&\lfloor \mathbf{PPtr} \rfloor && = \mathit{Loc} \\
&\lfloor \mathbf{PointsTo}_n\ \tau \rfloor && = \mathit{Loc} \times \lfloor \tau \rfloor \\
&\lfloor \mathbf{Dealloc}_n \rfloor && = \mathit{Loc} \\
&\lfloor \mathbf{PCell}_n\ \rfloor && = \mathit{CellId} \\
&\lfloor \mathbf{Cell::PointsTo}_n\ \tau \rfloor && = \mathit{CellId} \times \lfloor \tau \rfloor \\
&\lfloor \mathbf{LocalInvariant}(C, \tau, I) \rfloor && = \mathit{Name} \times C \\
&\lfloor \mathbf{AtomicInvariant}(C, \tau, I) \rfloor && = \mathit{Name} \times C \\
&\lfloor \mathbf{Resource}(\mathit{RA}) \rfloor && = \mathit{Name} \times M \quad \text{(see Figure 6.6)} \\
&\lfloor \mathbf{StorageResource}(\tau, \mathit{SP}) \rfloor && = \mathit{Name} \times P \quad \text{(see Figure 6.7)}
\end{aligned}
$$

Figure 6.3: **Context Interpretations and Type Interpretations.**

Essentially, this describes the local lifetime context, the typing context, and the invariant context before and after the instruction is executed. If the instruction returns a value, r, that value may be bound in the post-instruction typing context, $\mathbf{T}'$.

The "spec" part of the type-spec judgment is given by Φ, a *backward predicate transformer*, of sort:

$$(\lfloor \mathbf{T}' \rfloor \times \lfloor \mathbf{I}' \rfloor \to \mathit{Prop}) \to (\lfloor \mathbf{T} \rfloor \times \lfloor \mathbf{I} \rfloor \to \mathit{Prop})$$

The way to think about this is that we operate on predicates of the sort, $\lfloor \mathbf{T} \rfloor \times \lfloor \mathbf{I} \rfloor \to \mathit{Prop}$. Such a predicate can be understood as the condition under which it is safe to execute the code from a certain point; therefore, the predicate transformer Φ takes as input the conditions under which it is safe to execute the code starting right *after* the given instruction, and it returns the condition under which it is safe to execute the code starting right *before* the given instruction.

The input to each predicate is $\lfloor \mathbf{T} \rfloor \times \lfloor \mathbf{I} \rfloor$. The sort $\lfloor \mathbf{T} \rfloor$ is the product of the sorts of all the types, as shown in Figure 6.3. Of course, the inclusion of $\lfloor \mathbf{I} \rfloor$ in these predicates is specific to λ_{Verus}. We use $\lfloor \mathbf{I} \rfloor$ to track the invariants that are open as well as their associated constant parameters.

Note that these predicates do *not* operate on the lifetime contexts $\mathbf{E}$ or $\mathbf{L}$, even though they are otherwise key components of the typing rules. This allows specification-checking to be independent of lifetime-checking, a major advantage for the simplicity of the encoding and the engineering of the tools.[2]

Usually, the predicate transformer takes a form like:

$$\lambda \Psi, [\ldots]. \; \mathit{precondition} \wedge (\mathit{postcondition} \Rightarrow \Psi[\ldots])$$

With the ... placeholders representing variables from the pre-context and post-context. Often it may also have quantifiers, an existential for an operation that takes extra ghost parameters, and universal quantifiers for operations that return nondeterministic values:

$$\lambda \Psi, [\ldots]. \; \exists \_. \; \mathit{precondition} \wedge (\forall \_. \; \mathit{postcondition} \Rightarrow \Psi[\ldots])$$

For function bodies, we can define a similar type-spec judgment,

$$\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{I}; \mathbf{T} \vdash F \rightsquigarrow \Phi$$

where T is the return type of the function and:

$$\Phi : (\lfloor T \rfloor \to \mathit{Prop}) \to \lfloor \mathbf{T} \rfloor \times \lfloor I \rfloor \to \mathit{Prop}$$

The rules for typing function bodies are largely unsurprising, so for those I will just refer to prior work. Here, I will mostly be focusing on the Verus operations, wherein we will mostly be concerned with typing the instructions.

[2]Verus effectively throws away all lifetime information because it doesn't need it. I understand this is also true of Creusot, the tool that originally motivated RustHornBelt. Furtheremore, Linear Dafny was similar in this way as well, even though it had a much simpler type system: When we created Linear Dafny by modifying Dafny's type system, we did not need to modify its verification condition generation in any way.

Type-spec judgments for lifetimes and borrows

Figure 6.4 illustrates the basics of the borrowing and lifetime system. Let's start with "normal" (non-ghost) borrows. Lifetimes start and end with the **newlft** and **endlft** pseudo-instructions. During a **newlft**, we introduce a fresh lifetime variable α to the context, which is bounded by a vector of existing lifetimes of our choice.

Note that this does not immediately do anything else. To do something interesting, we can use a rule like C-BORROW-SHARED. This judgment, written with $\overset{\text{ctx}}{\Rightarrow}$ is a *type context inclusion*; it basically means we can replace the context on the left with the context on the right whenever we want. In this case it says that we can replace a normal owned pointer type:

$$p \lhd \mathbf{own}_n\ \tau$$

with two things. One, we get a shared reference:

$$p \lhd \&^{\alpha}_{\text{shr}}\tau \tag{6.1}$$

Two, we also get a judgment written with the $\lhd^{\dagger\alpha}$ notation:

$$p \lhd^{\dagger\alpha} \mathbf{own}_n\ \tau \tag{6.2}$$

The shared reference in (6.1) is usable as long as the lifetime α *is alive*. Liveness is given by the judgment $\mathbf{E}; \mathbf{L} \vdash \alpha$ alive which is used as the hypothesis for many rules involving shared references.

Meanwhile, the type given in (6.2) means that *after the lifetime expires*, we regain ownership of the τ. The rule for **endlft**, F-ENDLFT, is used to perform this "expiration." The $\mathbf{T} \Rightarrow^{\dagger\alpha} \mathbf{T}'$ judgment essentially just means that we replace each instance of $\lhd^{\dagger\alpha}$ with $\lhd$. Thus for example, (6.2) gets replaced by $p \lhd \mathbf{own}_n\ \tau$, which is what we had to begin with.

Manipulating shared borrows Now, there are a handful of things we can do with shared borrows while they're still alive. We can move a reference from a product to one of its fields, or to the occupant of a sum type, for example. You can also read a value from a shared reference when the type is copy. For physical shared references, this is all standard, and I again will simply refer to the prior work.

We are primarily interested in the ghost state references. The C-MOVE-BORROW-INSIDE-TRACKED shows that we can move a shared reference to "inside" the **Tracked** type so that we can "erase" the pointer.

Now, what are interesting things we can do with **Tracked** $\&^{\kappa}_{\text{trk}}\tau$? For this, we need to look at the special operations and special types.

Type-spec judgments for PPtr, PCell, and PointsTo

The Verus `PointsTo` introduced in Chapter 3 represents the permission to access a region of memory. Verus encodes it as a pair: a pointer and a value. In Figure 6.3, we see that $\lfloor \mathbf{PointsTo}_n\ \tau \rfloor = \mathit{Loc} \times \lfloor \tau \rfloor$. In λ_{Verus}, the *Loc* is the address type we use to index into memory.

Type-spec judgments on borrows and lifetimes
Starting and ending lifetimes

F-NEWLFT
$$\frac{\mathbf{E}; \mathbf{L}, \alpha \sqsubseteq_l \overline{\alpha} \mid \mathbf{K}; \mathbf{I}; \mathbf{T} \vdash F \rightsquigarrow \Psi}{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{I}; \mathbf{T} \vdash \mathbf{newlft}; F \rightsquigarrow \Psi}$$

F-ENDLFT
$$\frac{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{I}; \mathbf{T}' \vdash F \rightsquigarrow \Psi \quad \mathbf{T} \Rightarrow^{\dagger\alpha} \mathbf{T}'}{\mathbf{E}; \mathbf{L}, \alpha \sqsubseteq_l \overline{\alpha} \mid \mathbf{K}; \mathbf{I}; \mathbf{T} \vdash \mathbf{endlft}; F \rightsquigarrow \Psi}$$

Creating shared borrows

C-BORROW-SHARED
$$\mathbf{E}; \mathbf{L} \vdash p \lhd \mathbf{own}_n\, \tau \overset{\text{ctx}}{\Rightarrow} p \lhd \&^{\kappa}_{\text{shr}}\tau, p \lhd^{\dagger\kappa} \mathbf{own}_n\, \tau \rightsquigarrow \lambda\Psi[a], \Psi[a, a]$$

Manipulating borrows

C-MOVE-BORROW-INSIDE-TRACKED
$$\mathbf{E}; \mathbf{L} \vdash p \lhd \&^{\kappa}_{\text{shr}}\mathbf{Tracked}\, \tau \overset{\text{ctx}}{\Rightarrow} p \lhd \mathbf{Tracked}\, \&^{\kappa}_{\text{shr}}\tau, \rightsquigarrow \lambda\Psi[a], \Psi[a, a]$$

Lifetime inclusion and liveness

$$\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \mathbf{static} \qquad \frac{\kappa \sqsubseteq_l \overline{\kappa} \in \mathbf{L} \quad \kappa' \in \overline{\kappa}}{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq_l \kappa'} \qquad \frac{\kappa \sqsubseteq_e \kappa' \in \mathbf{E}}{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'} \qquad \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa$$

$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa' \quad \mathbf{E}; \mathbf{L} \vdash \kappa' \sqsubseteq \kappa''}{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa''}$$

$$\mathbf{E}; \mathbf{L} \vdash \mathbf{static}\ \text{alive} \qquad \frac{\kappa \sqsubseteq_l \overline{\kappa} \in \mathbf{L} \quad \forall i.\ \mathbf{E}; \mathbf{L} \vdash \overline{\kappa}_i\ \text{alive}}{\mathbf{E}; \mathbf{L} \vdash \kappa\ \text{alive}} \qquad \frac{\mathbf{E}; \mathbf{L} \vdash \kappa\ \text{alive} \quad \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E}; \mathbf{L} \vdash \kappa'\ \text{alive}}$$

Figure 6.4: **Selected type-specs for borrows and lifetimes.**

Type-spec judgments (PPtr and PointsTo)

$$\mathbf{E};\mathbf{L} \mid \ \vdash\ \mathbf{PPtr\_new}_n\ \dashv\ \mathsf{p}, \mathsf{pt}, \mathsf{d}.\ \mathsf{p} \lhd \mathbf{PPtr}\ ;\ \mathsf{pt} \lhd \mathbf{trk\ PointsTo}_n\ \lightning n\ ;\ \mathsf{d} \lhd \mathbf{trk\ Dealloc}_d$$
$$\rightsquigarrow \lambda\Psi.\ \forall p.\ \Psi[p, (p, ()), p]$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{p} \lhd \mathbf{PPtr}\ ;\ \mathsf{pt} \lhd \mathbf{trk\ PointsTo}_n\ \lightning n\ ;\ \mathsf{d} \lhd \mathbf{trk\ Dealloc}_n\ \vdash\ \mathbf{PPtr\_destroy}_n\ \mathsf{p}\ \dashv$$
$$\rightsquigarrow \lambda\Psi, [p_1, (p_2, ())].\ p_1 = p_2 \land \Psi[]$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{p} \lhd \mathbf{PPtr}\ ;\ \mathsf{v} \lhd \mathbf{own}_n\ \tau\ ;\ \mathsf{pt} \lhd \mathbf{trk\ PointsTo}_n\ \lightning n\ \vdash\ \mathbf{PPtr\_put}_n\ \mathsf{p}\ \mathsf{v}\ \dashv$$
$$\mathsf{pt}.\ \mathsf{pt} \lhd \mathbf{trk\ PointsTo}_n\ \tau$$
$$\rightsquigarrow \lambda\Psi, [p_1, v, (p_2, ())].\ p_1 = p_2 \land \Psi[(p_1, v)]$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{p} \lhd \mathbf{PPtr}\ ;\ \mathsf{v} \lhd \mathbf{own}_n\ \tau\ ;\ \mathsf{pt} \lhd \mathbf{PointsTo}_n\ \tau\ \vdash\ \mathbf{PPtr\_take}_n\ \mathsf{p}\ \dashv$$
$$\mathsf{v}, \mathsf{pt}.\ \mathsf{v} \lhd \tau\ ;\ \mathsf{pt} \lhd \mathbf{trk\ PointsTo}_n\ \lightning n$$
$$\rightsquigarrow \lambda\Psi, [p_1, (p_2, v)].\ p_1 = p_2 \land \Psi[(v, (p, ()))]$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{p} \lhd \mathbf{PPtr}\ ;\ \mathsf{pt} \lhd \&^{\kappa}_{\mathrm{trk}}\mathbf{PointsTo}_n\ \tau\ \vdash\ \mathbf{PPtr\_borrow}\ \mathsf{p}\ \dashv\ \mathsf{r}.\ \mathsf{r} \lhd \&^{\kappa}_{\mathrm{shr}}\tau$$
$$\rightsquigarrow \lambda\Psi, [p_1, (p_2, v)].\ p_1 = p_2 \land \Psi[v]$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{p} \lhd \mathbf{PPtr}\ ;\ \mathsf{i} \lhd \mathbf{int}\ ;\ \mathsf{pt} \lhd \mathbf{PointsTo}_n\ \mathbf{int}\ \vdash\ \mathbf{PPtr\_FAA}\ \mathsf{p}\ \mathsf{i}\ \dashv$$
$$\mathsf{j}, \mathsf{pt}.\ \mathsf{j} \lhd \mathbf{int}\ ;\ \mathsf{ptr} \lhd \mathbf{trk\ PointsTo}_n\ \mathbf{int}$$
$$\rightsquigarrow \lambda\Psi, [p, i, (p_2, j)].\ p = p_2 \land \Psi[(j, (p, i + j))]$$

Type-spec judgments (PCell and Cell::PointsTo)

$$\mathbf{E};\mathbf{L} \mid \ \vdash\ \mathbf{PCell\_new}_n\ \dashv\ \mathsf{c}, \mathsf{pt}.\ \mathsf{c} \lhd \mathbf{own}_n\ \mathbf{PCell}_n\ \tau\ ;\ \mathsf{pt} \lhd \mathbf{trk\ Cell::PointsTo}_n\ \lightning n$$
$$\rightsquigarrow \lambda\Psi.\ \forall\gamma.\ \Psi[\gamma, (\gamma, ())]$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{c} \lhd \mathbf{own}_n\ \mathbf{PCell}_n\ \tau\ ;\ \mathsf{pt} \lhd \mathbf{trk\ Cell::PointsTo}_n\ \lightning n\ \vdash\ \mathbf{PCell\_destroy}_n\ \mathsf{c}\ \dashv$$
$$\rightsquigarrow \lambda\Psi, [\gamma_1, (\gamma_2, ())].\ \gamma_1 = \gamma_2 \land \Psi[]$$

$$\frac{\mathbf{E};\mathbf{L} \vdash \kappa\ \mathsf{alive}}{\mathbf{E};\mathbf{L} \mid \mathsf{c} \lhd \&^{\kappa}_{\mathrm{shr}}\mathbf{PCell}_n\ \tau\ ;\ \mathsf{v} \lhd \mathbf{own}_n\ \tau\ ;\ \mathsf{pt} \lhd \mathbf{trk\ Cell::PointsTo}_n\ \lightning n\ \vdash\ \mathbf{PCell\_put}_n\ \mathsf{c}\ \mathsf{v}\ \dashv}$$
$$\mathsf{r}.\ \mathsf{r} \lhd \mathbf{trk\ Cell::PointsTo}_n\ \tau$$
$$\rightsquigarrow \lambda\Psi, [\gamma_1, v, (\gamma_2, ())].\ \gamma_1 = \gamma_2 \land \Psi[(\gamma_1, v)]$$

$$\frac{\mathbf{E};\mathbf{L} \vdash \kappa\ \mathsf{alive}}{\mathbf{E};\mathbf{L} \mid \mathsf{c} \lhd \&^{\kappa}_{\mathrm{shr}}\mathbf{PCell}_n\ \tau\ ;\ \mathsf{v} \lhd \mathbf{own}_n\ \tau\ ;\ \mathsf{pt} \lhd \mathbf{trk\ Cell::PointsTo}_n\ \tau\ \vdash\ \mathbf{PCell\_take}_n\ \mathsf{c}\ \dashv}$$
$$\mathsf{c}, \mathsf{pt}.\ \mathsf{c} \lhd \tau\ ;\ \mathsf{pt} \lhd \mathbf{Cell::PointsTo}_n\ \lightning n$$
$$\rightsquigarrow \lambda\Psi, [\gamma_1, (\gamma_2, v)].\ \gamma_1 = \gamma_2 \land \Psi[(v, (\gamma_1, ()))]$$

$$\frac{\mathbf{E};\mathbf{L} \vdash \kappa\ \mathsf{alive}}{\mathbf{E};\mathbf{L} \mid \mathsf{c} \lhd \&^{\kappa}_{\mathrm{shr}}\mathbf{PCell}_n\ \tau\ ;\ \mathsf{pt} \lhd \&^{\kappa}_{\mathrm{trk}}\mathbf{Cell::PointsTo}_n\ \tau\ \vdash\ \mathbf{PCell\_borrow}\ \mathsf{c}\ \dashv\ \mathsf{r}.\ \mathsf{r} \lhd \&^{\kappa}_{\mathrm{shr}}\tau}$$
$$\rightsquigarrow \lambda\Psi, [\gamma_1, (\gamma_2, v)].\ \gamma_1 = \gamma_2 \land \Psi[v]$$

Figure 6.5: **Selected type-specs for PPtr and PCell.**

Type-spec judgments (Resource Algebra Ghost State)
(*RA* is a monoid $(M, \cdot, \mathcal{V})$ satisfying the conditions in Figure 4.1.)

$$\lfloor \textbf{Resource}(RA) \rfloor = \mathit{Name} \times M$$

$$\mathbf{E};\mathbf{L} \mid \ \vdash \ \textbf{RA\_Alloc} \ \dashv \ \mathsf{r}.\ \mathsf{r} \lhd \textbf{trk Resource}(M) \rightsquigarrow \lambda\Psi.\ \exists m.\ \mathcal{V}(m) \land \forall \gamma.\ \Psi[(\gamma, m)]$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{g} \lhd \textbf{trk Resource}(M)\ ;\ \mathsf{h} \lhd \textbf{trk Resource}(M) \ \vdash \ \textbf{RA\_Join} \ \dashv \ \mathsf{r}.\ \mathsf{r} \lhd \textbf{trk Resource}(M)$$
$$\rightsquigarrow \lambda\Psi, [(g, \gamma_0), (h, \gamma_1)].\ \gamma_0 = \gamma_1 \land \Psi[(g \cdot h, \gamma_0)]$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{g} \lhd \textbf{trk Resource}(M) \ \vdash \ \textbf{RA\_Split} \ \dashv \ \mathsf{r},\mathsf{s}.\ \mathsf{r} \lhd \textbf{trk Resource}(M)\ ;\ \mathsf{s} \lhd \textbf{trk Resource}(M)$$
$$\rightsquigarrow \lambda\Psi, [(\gamma, g)].\ \exists f, h.\ (g = f \cdot h) \land \Psi[((\gamma, f), (\gamma, h)]$$

$$\mathbf{E};\mathbf{L} \mid \ \vdash \ \textbf{RA\_Unit} \ \dashv \ \mathsf{r}.\ \mathsf{r} \lhd \textbf{trk Resource}(M) \rightsquigarrow \lambda\Psi.\ \exists \gamma.\ \Psi[(\gamma, \epsilon)]$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{g} \lhd \&^{\kappa}_{\mathrm{trk}}\textbf{Resource}(M) \ \vdash \ \textbf{RA\_WeakenShared} \ \dashv \ \mathsf{r}.\ \mathsf{r} \lhd \&^{\kappa}_{\mathrm{trk}}\textbf{Resource}(M)$$
$$\rightsquigarrow \lambda\Psi, [(\gamma, g)].\ \exists h.\ (h \preccurlyeq g) \land \Psi[(\gamma, h)]$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{g} \lhd \&^{\kappa}_{\mathrm{trk}}\textbf{Resource}(M)\mathsf{h} \lhd \&^{\kappa}_{\mathrm{trk}}\textbf{Resource}(M) \ \vdash \ \textbf{RA\_JoinShared} \ \dashv \ \mathsf{r}.\ \mathsf{r} \lhd \&^{\kappa}_{\mathrm{trk}}\textbf{Resource}(M)$$
$$\rightsquigarrow \lambda\Psi, [(g, \gamma_1), (h, \gamma_2)].\ \gamma_1 = \gamma_2 \land (\forall f.\ (g \preccurlyeq f) \land (h \preccurlyeq f) \Rightarrow \Psi[(\gamma, f)])$$

$$\frac{\mathbf{E};\mathbf{L} \vdash \kappa \ \mathsf{alive}}{\mathbf{E};\mathbf{L} \mid \mathsf{g} \lhd \textbf{trk Resource}(M)\ ;\ \mathsf{h} \lhd \&^{\kappa}_{\mathrm{trk}}\textbf{Resource}(M) \ \vdash \ \textbf{RA\_Validate} \ \dashv \ \mathsf{r}.\ \mathsf{r} \lhd \textbf{trk Resource}(M)}$$
$$\rightsquigarrow \lambda\Psi, [(g, \gamma_1), (h, \gamma_2)].\ (\gamma_1 = \gamma_2) \land (\mathcal{V}(g \cdot h) \Rightarrow \Psi[(g, \gamma_1)])$$

$$\frac{\mathbf{E};\mathbf{L} \vdash \kappa \ \mathsf{alive}}{\mathbf{E};\mathbf{L} \mid \mathsf{g} \lhd \textbf{trk Resource}(M)\ ;\ \mathsf{h} \lhd \&^{\kappa}_{\mathrm{trk}}\textbf{Resource}(M) \ \vdash \ \textbf{RA\_Update} \ \dashv \ \mathsf{r}.\ \mathsf{r} \lhd \textbf{trk Resource}(M)}$$
$$\rightsquigarrow \lambda\Psi, [(g, \gamma_1), (h, \gamma_2)].\ \exists B.\ (g \cdot h \rightsquigarrow B \cdot h) \land (\forall b.\ b \in B \Rightarrow \Psi[(b, \gamma_1)])$$

(where $B \cdot h \triangleq \{b \cdot h : b \in B\}$)

Figure 6.6: **Selected type-specs for RA-based ghost state.**

Type-spec judgments (Storage Protocol Ghost State)
(*SP* is a storage protocol with protocol monoid P and storage monoid $(\mathbf{map}\ K\ \lfloor\tau\rfloor) \cup \{\lightning\}$ satisfying the conditions in Figure 4.9a.)

$$\lfloor \mathbf{StorageResource}(SP) \rfloor = \mathit{Name} \times P$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{p} \lhd \&^{\kappa}_{\mathrm{trk}}\mathbf{StorageResource}(M) \vdash \mathbf{SP\_Guard} \dashv \mathsf{s}.\ \mathsf{s} \lhd \&^{\kappa}_{\mathrm{trk}}\tau \rightsquigarrow \lambda\Psi, [(p,\gamma_1)].\ \exists k, s.\ (p \twoheadrightarrow [k \mapsto s]) \land \Psi[s]$$

$$\frac{\mathbf{E};\mathbf{L} \vdash \kappa\ \mathsf{alive}}{\begin{array}{c}\mathbf{E};\mathbf{L} \mid \mathsf{p} \lhd \mathbf{trk\ StorageResource}(M)\ ;\ \mathsf{x} \lhd \&^{\kappa}_{\mathrm{trk}}\mathbf{StorageResource}(M)\ ;\ \mathsf{s} \lhd \tau \\ \vdash \mathbf{SP\_Exchange\_With\_Shared} \dashv \\ \mathsf{q}, \mathsf{t}.\ \mathsf{q} \lhd \mathbf{trk\ StorageResource}(M)\ ;\ \mathsf{t} \lhd \tau \\ \rightsquigarrow \lambda\Psi, [(p,\gamma_1),(x,\gamma_2),v_1].\ \exists k_1, k_2, q, v_2.\ (\gamma_1 = \gamma_2) \land ((p \cdot x, [k_1 \mapsto v]) \dot{\rightsquigarrow} (q \cdot x, [k_2 \mapsto v_2])) \land \Psi[(q,\gamma_1), v_2]\end{array}}$$

Figure 6.7: **Selected type-specs for Storage Protocol-based ghost state.** *Many rules are similar to those for ghost RA state, so we include the most unique ones here. Operations are simplified to only operate on singleton "elements" instead of arbitrary maps.*

Type-spec judgments (Invariants)

$$\mathbf{E};\mathbf{L} \mid \mathsf{t} \lhd \tau \vdash \mathbf{Local\_New} \dashv \mathsf{r}.\ \mathsf{r} \lhd \mathbf{LocalInvariant}(C, \tau, I) \rightsquigarrow \lambda\Psi, [a, \mathcal{E}_{user}].\ \exists (c : C)(\gamma : \mathit{Name}).\ I(c,a) \land \Psi[(\gamma, c), \mathcal{E}_{user}]$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{i} \lhd \mathbf{LocalInvariant}(C, \tau, I) \vdash \mathbf{Local\_Destroy} \dashv \mathsf{r}.\ \mathsf{r} \lhd \tau \rightsquigarrow \lambda\Psi, [(\gamma, c), \mathcal{E}_{user}].\ (\gamma \in \mathcal{E}_{user}) \land \forall a.\ I(c,a) \Rightarrow \Psi[a, \mathcal{E}_{user}]$$

$$\frac{\mathbf{E};\mathbf{L} \vdash \kappa\ \mathsf{alive}}{\mathbf{E};\mathbf{L} \mid \mathsf{i} \lhd \&^{\kappa}_{\mathrm{shr}}\mathbf{LocalInvariant}(C, \tau, I) \vdash \mathbf{Local\_Open} \dashv \mathsf{r}, \iota.\ \mathsf{r} \lhd \tau\ ;\ \iota \lhd \mathbf{InLocal}(C, \tau, I) \rightsquigarrow \lambda\Psi, [(\gamma, c), \mathcal{E}_{user}].\ (\gamma \in \mathcal{E}_{user}) \land \forall a.\ I(c,a) \Rightarrow \Psi[a, (\gamma, c), \mathcal{E}_{user} \backslash \{\gamma\}]}$$

$$\mathbf{E};\mathbf{L} \mid \mathsf{t} \lhd \tau\ ;\ \iota \lhd \mathbf{InLocal}(C, \tau, I) \vdash \mathbf{Local\_Close} \dashv \iota \lhd \mathbf{InLocal}(C, \tau, I) \rightsquigarrow \lambda\Psi, [a, (\gamma, c), \mathcal{E}_{user}].\ I(c,a) \land \Psi[\mathcal{E}_{user} \cup \{\gamma\}]$$

Figure 6.8: **Type-specs for invariant operations.** *The type-specs for the* ***AtomicInvariant*** *operations are identical.*

Notably, `PointsTo` can point to either initialized or uninitialized memory, so I should point out the version here is a bit different; the **PointsTo**$_n\ \tau$ always points to an initialized τ, so we instead use **PointsTo**$_n\ \lightning_n$ for uninitialized points-to. Observe, for example, how the type-specs for **PPtr_put** (Figure 6.5) and **PPtr_take** swap back and forth between **PointsTo**$_n\ \tau$ and **PointsTo**$_n\ \lightning_n$. To recover the behavior of the real Verus type, we could use an enum over **PointsTo**$_n\ \tau$ and **PointsTo**$_n\ \lightning_n$.

There is no reason to enumerate every possible atomic operation, so the figure lists just one: the operation for **PPtr_FAA**, i.e., the atomic fetch-and-add.[3] As we can see, at this point it isn't meaningfully different than **PPtr_take** and **PPtr_put**. However, recall our earlier discussion about the semantics of λ_{Verus}, atomic operations, and the global atomic lock. This argument would apply to something like **PPtr_FAA**, which is atomic in the operational semantics of λ_{Verus}, but it would not apply to something like **PPtr_put**, which uses non-atomic memory ordering and thus is not atomic in the operational semantics.

Type-spec judgments for Resource Algebras and Storage Protocols

Judgments for RA-based and storage protocol-based ghost state are shown in Figure 6.6 and Figure 6.7. These are translations of the operations from the Verus interfaces in Chapter 4. Since we already did the hard work of motivating what we want these specifications to be, there isn't much more to say about them now.

Since the storage protocol rules are so hefty, I did my best to focus Figure 6.7 on the important bits. The two rules presented are the two main unique features of storage protocols: guarding and exchanging. I used the deterministic version of exchange. Also, in order to avoid having to spell out a variable-size container type, I present a special case with singleton elements instead of maps.

Type-spec judgments for invariants

The invariants have complicated specifications because of their interactions with the invariant context **I**.

The interpretation of the the invariant context contains the following information:

- For each open invariant, the *name* associated with it and the constant $c : C$ associated with it.
- A *mask* $\mathcal{E}_{user} : \mathcal{P}(Name)$, that is, a set of names that represents invariants that may be opened (i.e., the complement of the set of invariants that are currently open). We call this the "user mask."

Inspecting the type-specs in Figure 6.8, we see that opening an invariant (**Local_Open**) requires us to check that its name (γ) is not in $\mathcal{E}_{user}$, and it also removes γ from $\mathcal{E}_{user}$ for the post-instruction invariant context. Meanwhile, closing an invariant (**Local_Close**) returns γ to the $\mathcal{E}_{user}$. Also observe that **Local_Close** requires the invariant predicate I to be met, using the value of c that is tracked in the invariant context.

[3]The Verus primitives (mirroring Rust's standard library) use different types for atomic operations, but at the formal level, it makes sense to use the same types all memory operations, both atomic and non-atomic.

Local_New and **Local_Destroy** are more straightforward, though there is one small detail worth commenting on. Initially, I was planning to forego **Local_Destroy**'s requirement that $\gamma \in m$, which prevents an unsoundness due to the possibility of destroying a **LocalInvariant** while it was open. Instead, the plan was to prevent this unsoundness via lifetime-checking: specifically, in **Local_Open**, I would force the lifetime of the borrow to extend to the **Local_Close** instruction. However, in the process of constructing the formal model present in this chapter, I identified a problem with this scheme (and a relevant soundness issue in Verus[4]). Namely, it turned out that it was possible for the program to be non-terminating, preventing **Local_Close** from ever being reached, thus failing to force the lifetime to be extended. I still believe this could be made to work with some tweaks, but on reflection, I realized it was simpler to remove the restriction entirely and instead add the $\gamma \in \mathcal{E}_{user}$ precondition. This was not only simpler from a formal perspective, but also from a user perspective as well; in the past, the lifetime restriction caused complications in some scenarios that demanded awkward workarounds. By making this change, I was able to cut the awkward workarounds entirely.

6.3.4 Marker traits

Figure 6.9 shows the derivations of marker traits for `Copy`, `Send`, and `Sync`, in accordance with Figure 3.6.

6.3.5 Subtyping

Our subtyping rules are shown in Figure 6.10. The subtyping judgment is $\mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow_f \tau_2$. The function $f : \lfloor \tau_1 \rfloor \to \lfloor \tau_2 \rfloor$ is technically needed so we can transform the type-specs when replacing something in the context with a subtype; in practice, this function is always a trivial isomorphism, though.

The main nontrivial subtyping relationship in Rust has to do with lifetime inclusion. Specifically, when $\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'$, we have that $\&^{\kappa'}_{\text{shr}}\tau$ is a subtype of $\&^{\kappa}_{\text{shr}}\tau$ (Subtype-bor-lft). This makes sense since κ' outlives κ; therefore if we have a reference is valid throughout κ', then it will be valid through κ.

The remaining rules in our figure relate to the variances of various type constructors. Most type constructors are covariant.

Notably, **StorageResource**(τ, SP) is non-variant in τ (the type of the stored ghost state). This actually seems to be somewhat over-conservative, and as we will discuss more later (§9.4.3), this actually makes it difficult to construct a type like `Rc<T>` or `Arc<T>` that is correctly covariant in `T`. However, it is not obvious how to cleanly design a rule that is more precise. We leave this as an open question.

6.3.6 Recursive Types

In λ_{Rust}, recursive types $\mu T.\ \tau$ are allowed provided every occurrence of T in τ is behind a pointer. This restriction is necessary to avoid unrepresentable types. In λ_{Verus}, we inherit this restriction.

[4] https://github.com/verus-lang/verus/issues/1102

Marker traits: Send, Sync, and Copy

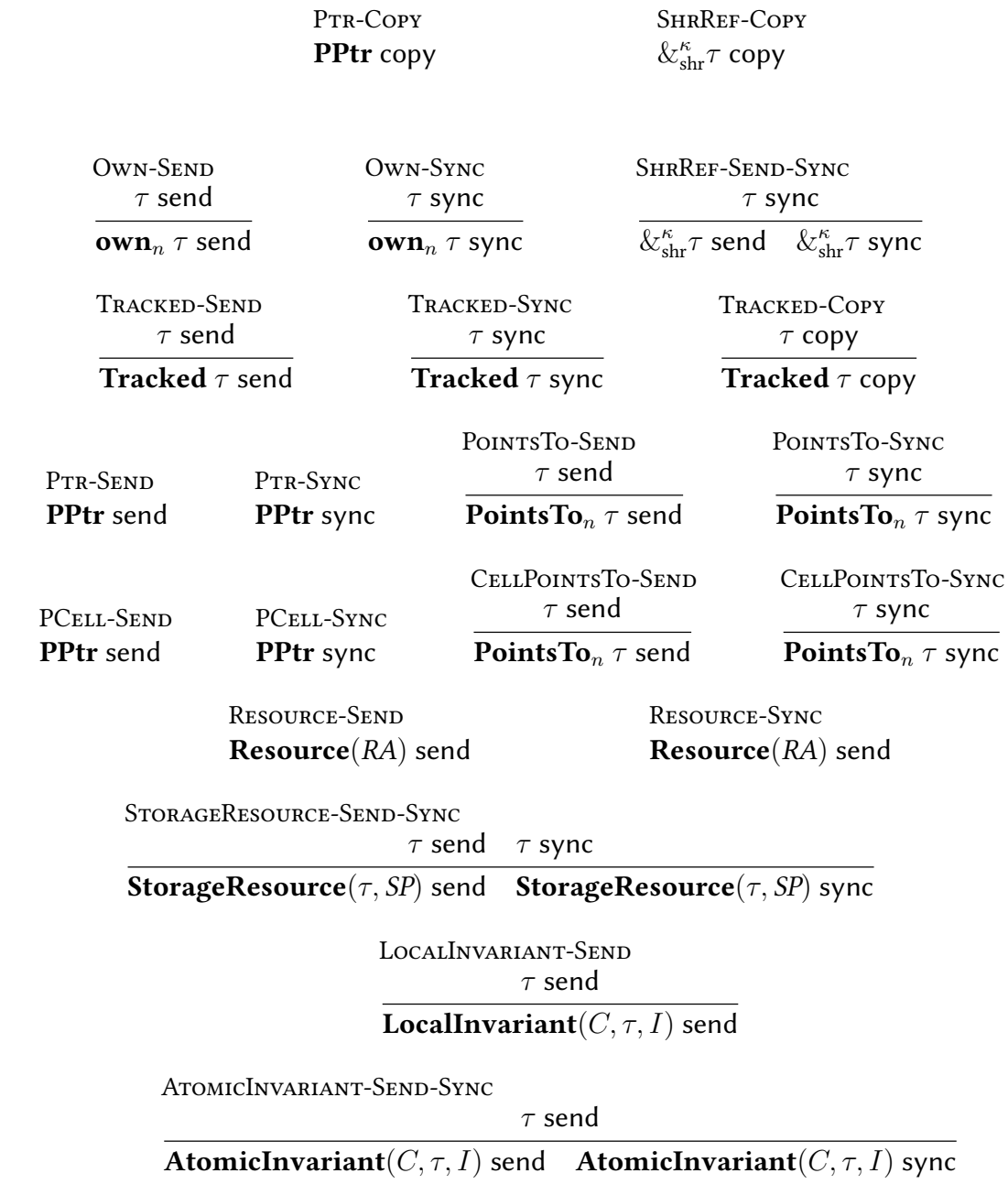


Figure 6.9: **Marker traits (Copy, Send, and Sync) for selected types in the λ_{Verus} type system.**

Subtyping

$$\textsc{Subtype-bor-lft}\ \frac{\mathbf{E};\mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E};\mathbf{L} \vdash \&^{\kappa}_{\mathrm{shr}}\tau \Rightarrow_{\mathsf{id}} \&^{\kappa'}_{\mathrm{shr}}\tau} \qquad \textsc{Subtype-own}\ \frac{\mathbf{E};\mathbf{L} \vdash \tau_1 \Rightarrow_f \tau_2}{\mathbf{E};\mathbf{L} \vdash \mathbf{own}_n\ \tau_1 \Rightarrow_f \mathbf{own}_n\ \tau_2} \qquad \textsc{Subtype-bor-shr}\ \frac{\mathbf{E};\mathbf{L} \vdash \tau_1 \Rightarrow_f \tau_2}{\mathbf{E};\mathbf{L} \vdash \&^{\kappa}_{\mathrm{shr}}\tau_1 \Rightarrow_f \&^{\kappa}_{\mathrm{shr}}\tau_2}$$

$$\textsc{Subtype-Tracked}\ \frac{\mathbf{E};\mathbf{L} \vdash \tau_1 \Rightarrow_f \tau_2}{\mathbf{E};\mathbf{L} \vdash \mathbf{Tracked}\ \tau_1 \Rightarrow_f \mathbf{Tracked}\ \tau_2} \qquad \textsc{Subtype-PointsTo}\ \frac{\mathbf{E};\mathbf{L} \vdash \tau_1 \Rightarrow_f \tau_2 \qquad \forall \ell, v.\ f'((\ell, v)) = (\ell, f(v))}{\mathbf{E};\mathbf{L} \vdash \mathbf{PointsTo}_n\ \tau_1 \Rightarrow_{f'} \mathbf{PointsTo}_n\ \tau_2}$$

$$\textsc{Subtype-CellPointsTo}\ \frac{\mathbf{E};\mathbf{L} \vdash \tau_1 \Rightarrow_f \tau_2 \qquad \forall \gamma, v.\ f'((\gamma, v)) = (\gamma, f(v))}{\mathbf{E};\mathbf{L} \vdash \mathbf{Cell::PointsTo}_n\ \tau_1 \Rightarrow_{f'} \mathbf{Cell::PointsTo}_n\ \tau_2}$$

Figure 6.10: **Selected subtyping rules.** *In addition to the ones shown, there are trivial rules about products, sums, recursive types, and so on.*

However, in λ_{Verus}, there is an additional restriction that deserves attention. In particular, the representation sort $\lfloor \mu T.\ \tau \rfloor$ needs to be constructible in the meta-logic, which again, we take to be the Calculus of Inductive Constructions. As stated earlier, Verus's specification language is formalized by a lowering into CoIC, so this restriction does in fact correspond to what Verus will accept or reject.

In CoIC, we can construct $\lfloor \mu T.\ \tau \rfloor$ as long as $\lfloor T \rfloor$ never appears in a negative position in $\lfloor \tau \rfloor$. It is worth noting that the **LocalInvariant**(C, τ, I) and **AtomicInvariant**(C, τ, I) types have been carefully designed to facilitate the satisfaction of this condition. To explain how, allow me to first explain how a more naive design runs into severe restrictions.

The naive design In earlier versions of Verus, the **LocalInvariant** and **AtomicInvariant** types were designed in a naive way that prevented the construction of recursive types that involved invariant types. Specifically, in these early versions, the type constructors each took only a single parameter, τ.

Upon construction of an invariant object, the user would supply a predicate describing the allowed values of τ. As a result, we had (in effect):

$$\lfloor \mathbf{LocalInvariant}(\tau) \rfloor \triangleq \mathit{Name} \times (\lfloor \tau \rfloor \to \mathit{Prop})$$

We immediately see the problem: $\lfloor \tau \rfloor$ appears in a negative position. This would make it impossible to handle Challenge RC-3.

The solution Resolving this problem was the primary motivation for factoring the invariant predicate into $c : C$ and $I : C \times \lfloor \tau \rfloor \to \mathit{Prop}$. Now I is fixed at the static type level and only C appears in the representation sort:

$$\lfloor \mathbf{LocalInvariant}(\tau) \rfloor \triangleq \mathit{Name} \times C$$

It may not be entirely obvious that this actually solves the problem, rather than simply smuggling the negative-use to some other problematic location. Therefore, it is worth working out an example.

Example Consider a user-defined type like:

$$\begin{aligned}\textbf{PtrWithInv}_n(\tau, \mathit{wf}) \triangleq\ & \textbf{PPtr} \times \textbf{Tracked}\ (\textbf{LocalInvariant}(\mathit{Loc}, I_{\text{loc}}(\mathit{wf}), (\textbf{PointsTo}_n\ \tau))) \\ & \text{where} \\ I_{\text{loc}} \triangleq\ & \lambda \mathit{wf}, \ell, \mathit{points\_to}.\ \mathit{points\_to}.1 = \ell \wedge \mathit{wf}(\mathit{points\_to}.2)\end{aligned}$$

Effectively, $\textbf{PtrWithInv}_n$ has a pointer and an invariant containing the pointer permission. The invariant is configured with a predicate that says the *points_to* has a specific pointer, and it also ensures that the value being pointed to obeys some user-specified predicate $\mathit{wf} : \lfloor \tau \rfloor \to \mathit{Prop}$. This type has its own well-formed predicate:

$$\begin{aligned}&\mathit{wf}_{\mathit{PtrWithInv}} : \lfloor \textbf{PtrWithInv}_n(\tau, \mathit{wf}) \rfloor \to \mathit{Prop} \\ &\mathit{wf}_{\mathit{PtrWithInv}}((p, (\gamma, c))) \triangleq (p = c)\end{aligned}$$

This just says that the actual physical pointer matches the pointer that the invariant is configured to. Now, suppose we wanted to instantiate this type with itself to create, e.g., a linked list.

$$\textbf{LinkedList}_n(\tau) \triangleq \mu T.\ \textbf{Option PtrWithInv}_n(T, \mathit{ifSome}\ \mathit{wf}_{\mathit{PtrWithInv}})$$

Where $\mathit{ifSome}\ \mathit{wf}_{\mathit{PtrWithInv}}$ applies $\mathit{wf}_{\mathit{PtrWithInv}}$ to an optional value.

The interpretation sort of this type is:

$$\textbf{Option}\ (\mathit{Loc} \times (\mathit{Name} \times \mathit{Loc}))$$

Note that $\lfloor T \rfloor$ does not appear at all in this type, let alone in a negative position. However, we were still able to specify all the necessary invariants via predicates in the type constructors.

6.4 Soundness of λ_{Verus} specifications

Continuing to follow RustBelt and RustHornBelt, we will sketch how to prove the soundness of the specifications using *logical type soundness* [74]. Logical type soundness is an approach to semantic type soundness wherein we define *semantic interpretations*, denoted by $[\![\cdot]\!]$, for all judgments in the type system as *Iris* propositions. The result is that all typing rules can be interpreted as *theorems in Iris* which can then be proved. Given the already close connection between Verus primitive types and Iris, this is naturally an appealing approach to take.

The two main theorems are:

Theorem 3 (Fundamental theorem of logical relations) *For any inference rule, if we wrap all judgments in $[\![\cdot]\!]$, then the resulting Iris theorem holds.*

Theorem 4 (Adequacy) *Let f be a function where*

$$[\![\emptyset; \emptyset \mid \emptyset; \emptyset \vdash \ f \ \dashv \ x.\, x \lhd \textbf{\textit{fn}}() \to (); \emptyset \rightsquigarrow \lambda \Psi.\, \Psi]\!]$$

Then when we execute f, passing it a trivial continuation, no execution ends in a stuck state.

Observe how the two theorems can be applied together. If f typechecks, then by Theorem 3, the semantic interpretation $[\![\cdot]\!]$ of that judgment holds; then by Theorem 4, f does not get stuck.

Theorem 4 basically follows from Iris's adequacy theorem. Thus the bulk of the work is in proving Theorem 3, both the creative work of designing the semantic interpretations $[\![\cdot]\!]$, and the grunt work of checking that every judgment holds in the resulting semantic model. In what follows, I will cover the semantic definitions of all the important λ_{Verus} types and outline a representative subset of the proofs for key judgments.

A novel feature of our approach for λ_{Verus} is that we are going to handle all shared references in terms of Leaf's $\twoheadrightarrow$ operator. This has the advantage that all of the Leaf-based ghost state laws will follow fairly straightforwardly. However, by entirely rethinking the way shared references work, it does mean we need to rework one of the cornerstone pillars of RustBelt: its *lifetime logic*.

6.4.1 The Leaf Lifetime Logic

The *Lifetime Logic* is a small "library" proved in Iris used to help build the semantic models of borrow types and other aspects related to lifetimes. Our version of the Lifetime Logic, the *Leaf Liftetime Logic*, is presented in Figure 6.11. In §6.4.2 we will construct a model of the Leaf Lifetime Logic to prove its soundness.

There are two main differences between the Leaf Lifetime Logic and the original:

- Ours uses the Leaf "guards" operator ($\twoheadrightarrow$).
- Ours does not account for "full borrows" (which would be needed for mutable references).

The Leaf Lifetime Logic, like the original, begins with an algebra of *lifetimes*, usually denoted by κ. Lifetimes form a monoid, with a unit ϵ, the lifetime that is always active, and intersection $\kappa \sqcap \kappa'$, which effectively represents "the lifetime that is active while both constituent lifetimes are active."

The Leaf Lifetime Logic
Propositions: $[\kappa] \quad [\dagger\kappa] \quad \kappa \sqsubseteq \kappa'$
(where κ, κ' : *Lifetime*)

LLftL-Begin
$\mathsf{True} \Rrightarrow_{\mathcal{N}_{lft}} \exists \kappa.\, [\kappa] * ([\kappa] \Rrightarrow_{\mathcal{N}_{lft}} [\dagger\kappa])$

LLftL-Not-Own-End
$[\kappa] * [\dagger\kappa] \vdash \mathsf{False}$

LLftL-BorrowShared
$\triangleright P \Rrightarrow_{\mathcal{N}_{lft}} ([\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} \triangleright P) * ([\dagger\kappa] \Rrightarrow_{\mathcal{N}_{lft}} \triangleright P)$

LLftL-Incl-Isect
$(\kappa \sqcap \kappa') \sqsubseteq \kappa$

LLftL-Incl-Glb
$(\kappa \sqsubseteq \kappa') * (\kappa \sqsubseteq \kappa'') \vdash (\kappa \sqsubseteq (\kappa' \sqcap \kappa''))$

LLftL-Tok-Inter
$[(\kappa \sqcap \kappa')] \dashv\vdash [\kappa] \wedge [\kappa']$

LLftL-End-Inter
$[\dagger(\kappa \sqcap \kappa')] \dashv\vdash [\dagger\kappa] \vee [\dagger\kappa']$

LLftL-Tok-Unit
$\mathsf{True} \vdash [\epsilon]$

LLftL-End-Unit
$[\dagger\epsilon] \vdash \mathsf{False}$

LLftL-Incl-Dead-Implies-Dead
$([\kappa] \sqsubseteq [\kappa']) * [\dagger\kappa'] \Rrightarrow_{\mathcal{N}_{lft}} [\dagger\kappa]$

LLftL-Dead-Persistent
$\mathsf{Persistent}([\dagger\kappa])$

$$\kappa \sqsubseteq \kappa' \triangleq [\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} [\kappa']$$

Figure 6.11: **The Leaf Lifetime Logic.** *Naming convention: names correspond to the closest analogous rule in RustBelt's original lifetime logic [33]. "LLftL" stands for "Leaf Lifetime Logic."*

Lifetime tokens The Leaf Lifetime Logic establishes a proposition $[\kappa]$ meaning "the lifetime κ is active" and another proposition $[\dagger\kappa]$ meaning "the lifetime κ has expired." Once a lifetime has expired, it will always be expired, so $[\dagger\kappa]$ is a persistent proposition. Observe that a lifetime intersection $\kappa \sqcap \kappa'$ is active iff both κ and κ' are active (LLftL-Tok-Inter) the intersection is dead if either κ or κ' is dead (LLftL-End-Inter).

Starting and ending a lifetime LLftL-Begin allows the user to start a lifetime. They obtain a new token $[\kappa]$, indicating that κ is alive for some fresh κ, and they also receive the ability to end the lifetime at any point by exchanging the $[\kappa]$ token for a $[\dagger\kappa]$ token.

Tying propositions to lifetimes At any point, the user can apply LLftL-BorrowShared to relinquish ownership of some proposition P. In exchange, they get two things:

- The *borrow* $[\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} \triangleright P$ which gives them shared access to P as long as κ is alive, and
- The *expiration* $[\dagger\kappa] \mathrel{\Rightarrow\!\!\!\!\ast}_{\mathcal{N}_{lft}} \triangleright P$, that is, the ability to get ownership of P back once κ is expired. (Recall that a $\mathrel{\Rightarrow\!\!\!\!\ast}$ proposition can only be used once.)

Lifetime inclusion The last proposition introduced by the lifetime logic is the *lifetime inclusion*, $\kappa \sqsubseteq \kappa'$, which by definition, is just $\kappa \twoheadrightarrow_{\mathcal{N}_{lft}} \kappa'$. This persistent proposition essentially means that as long as κ is alive, κ' is as well. Note that the Iris proposition $\kappa \sqsubseteq \kappa'$ is actually stronger that simply saying κ is contained in κ' in the monoidal lifetime algebra, since $\twoheadrightarrow$ propositions can be constructed in a number of ways. Also observe that if we have an inclusion $\kappa \sqsubseteq \kappa'$ and a borrow $[\kappa'] \twoheadrightarrow_{\mathcal{N}_{lft}} P$ then we can easily combine (Guard-Trans) to get $[\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} P$.

Finally, the lifetime logic gives us the ability to deduce that if $\kappa \sqsubseteq \kappa'$ and κ' has expired, then κ must have expired (LLftL-Incl-Dead-Implies-Dead).[5]

A note on notation The $\mathcal{N}_{lft}$ namespace is needed for invariants used in constructing the model of the Leaf Lifetime Logic. For the rest of this section, we will just write $\twoheadrightarrow$, eliding the mask annotation to avoid clutter, leaving the $\mathcal{N}_{lft}$ namespace assumed. We will also elide 'later' annotations as in $\twoheadrightarrow^{\triangleright n}$, but we will return to this point in §6.4.9.

Lifetime contexts The reason for introducing the Leaf Lifetime Logic is so that we can provide semantic interpretations of lifetime variables, lifetime contexts, and shared references. The semantic interpretations of lifetime contexts are given in Figure 6.12.

Let us run through an example of using the Leaf Lifetime Logic to prove semantic soundness of a simple lifetime-related rule. The rule we consider is one from λ_{Rust}:

$$\text{F-equalize}\ \frac{\mathbf{E}, \alpha \sqsubseteq_e \kappa, \kappa \sqsubseteq_e \alpha; \mathbf{L} \mid \mathbf{K}; \mathbf{I}; \mathbf{T} \vdash F \rightsquigarrow \Psi}{\mathbf{E}; \mathbf{L}, \alpha \sqsubseteq_l [\kappa] \mid \mathbf{K}; \mathbf{I}; \mathbf{T} \vdash F \rightsquigarrow \Psi}$$

[5] In RustBelt's classic lifetime logic, the equivalent of this rule immediately followed from the definition of $\sqsubseteq$. Our definition of $\sqsubseteq$ is different, and this part does not immediately follow, and thus it needs to be an explicit rule in the Leaf Lifetime Logic.

This rule basically says, "if α is bounded by κ, but has no other constraints, then we can collapse the two lifetimes." To prove semantic soundness, it suffices to prove this entailment in Iris:

$$[\![\mathbf{E}, \alpha \sqsubseteq_e \kappa, \kappa \sqsubseteq_e \alpha; \mathbf{L} \mid \mathbf{K}; \mathbf{I}; \mathbf{T} \vdash F \rightsquigarrow \Psi]\!] \vdash [\![\mathbf{E}; \mathbf{L}, \alpha \sqsubseteq_l [\kappa] \mid \mathbf{K}; \mathbf{I}; \mathbf{T} \vdash F \rightsquigarrow \Psi]\!]$$

And for this, it suffices to show:

$$[\![\alpha \sqsubseteq_l [\kappa]]\!] \Rrightarrow_{\mathcal{N}_{lft}} [\![\alpha \sqsubseteq_e \kappa]\!] * [\![\kappa \sqsubseteq_e \alpha]\!]$$

Now, $[\![\alpha \sqsubseteq_l [\kappa]]\!]$ gives us $[\kappa']$ such that $[\![\alpha]\!] = [\![\kappa]\!] \sqcap \kappa'$. We want both $[[\![\alpha]\!]] \twoheadrightarrow_{\mathcal{N}_{lft}} [[\![\kappa]\!]]$ and $[[\![\kappa]\!]] \twoheadrightarrow_{\mathcal{N}_{lft}} [[\![\alpha]\!]]$. The first is immediate (LLftL-Incl-Isect). For the second, we can eternalize $[\kappa']$ via Guard-Forever to get True $\twoheadrightarrow_{\mathcal{N}_{lft}} [\kappa']$, implying $[[\![\kappa]\!]] \twoheadrightarrow_{\mathcal{N}_{lft}} [\kappa']$. Then the result follows from LLftL-Incl-Glb.

6.4.2 A model of the Leaf Lifetime Logic

We can model lifetimes κ as finite subsets of $\mathbb{N}$. We define lifetime intersection to be set union:

$$\kappa \sqcap \kappa' \triangleq \kappa \cup \kappa'$$
$$\epsilon \triangleq \emptyset$$

The idea is that individual elements $k \in \mathbb{N}$ can be "alive" or "dead" (or unused) and a lifetime κ is consider to be alive iff all of its elements $k \in \kappa$ are alive.

We can create a resource with the following properties, assuming some global singleton resource $\mathsf{LtState}(A, D)$, where A and D are finite subsets of $\mathbb{N}$ representing the alive elements and dead elements respectively.

$$(k \notin A \wedge k \notin D) * \mathsf{LtState}(A, D) \Rrightarrow \mathsf{LtState}(A \cup \{k\}, D) * \mathsf{Alive}(k) * \mathsf{Alive}(k) \quad (6.3)$$
$$\mathsf{LtState}(A, D) \wedge \mathsf{Alive}(k) \vdash k \in A \quad (6.4)$$
$$\mathsf{LtState}(A, D) \wedge \mathsf{Dead}(k) \vdash k \in D \quad (6.5)$$
$$\mathsf{Alive}(k) \wedge \mathsf{Dead}(k) \Rrightarrow \mathsf{False} \quad (6.6)$$
$$\mathsf{LtState}(A \cup \{k\}, D) * \mathsf{Alive}(k) * \mathsf{Alive}(k) \Rrightarrow \mathsf{LtState}(A, D \cup \{k\}) * \mathsf{Dead}(k) \quad (6.7)$$
$$\mathsf{persistent}(\mathsf{Dead}(k)) \quad (6.8)$$
$$(d \in D) * \mathsf{LtState}(A, D) \vdash \mathsf{Dead}(k) \quad (6.9)$$
$$\mathsf{LtState}(A, D) \vdash A \cap D = \emptyset \quad (6.10)$$

The reason we need two "copies" of the Alive token will become clearer below.

Now, we assume we have this guard in context:

$$\mathsf{True} \twoheadrightarrow_{\mathcal{N}_{lft}} \left(\exists A, D.\ \mathsf{LtState}(A, D) * \mathop{\Large *}_{a \in A} \mathsf{Alive}(a)\right) \quad (6.11)$$

Then we define:

$$[\kappa] \triangleq \mathop{\Large *}_{k \in \kappa} \mathsf{Alive}(k)$$
$$[\dagger\kappa] \triangleq \exists k.\ (k \in \kappa) * \mathsf{Dead}(k)$$

Semantic models of contexts and judgments

$$
\begin{aligned}
[\![\tau \text{ copy}]\!] &\triangleq \forall a, \mathit{tid}, \overline{v}.\ [\![\tau]\!].\mathrm{ghost}(a, \mathit{tid}) \Rrightarrow [\![\tau]\!].\mathrm{ghost}(a, \mathit{tid}) * [\![\tau]\!].\mathrm{ghost}(a, \mathit{tid}) \\
[\![\tau \text{ send}]\!] &\triangleq \forall a, \mathit{tid}, \mathit{tid}', \overline{v}.\ ([\![\tau]\!].\mathrm{ghost}(a, \mathit{tid}) \Rrightarrow [\![\tau]\!].\mathrm{ghost}(a, \mathit{tid}')) \\
&\qquad * ([\![\tau]\!].\mathrm{phys}(a, \mathit{tid}) = [\![\tau]\!].\mathrm{phys}(a, \mathit{tid}')) \\
[\![\tau \text{ sync}]\!] &\triangleq \forall a, \mathit{tid}, \mathit{tid}', \overline{v}, G.\ (G \twoheadrightarrow_{\mathcal{N}_{lft}} [\![\tau]\!].\mathrm{ghost}(a, \mathit{tid}, \overline{v})) \Rrightarrow \\
&\qquad (G \twoheadrightarrow_{\mathcal{N}_{lft}} [\![\tau]\!].\mathrm{ghost}(a, \mathit{tid}', \overline{v})) \\
&\qquad\qquad * ([\![\tau]\!].\mathrm{phys}(a, \mathit{tid}) = [\![\tau]\!].\mathrm{phys}(a, \mathit{tid}'))
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathbf{T}]\!](\overline{a}, \mathit{tid}) &\triangleq \mathop{\Large *}_{(t,a)\in(\mathbf{T},\overline{a})} [\![t]\!](a, \mathit{tid}) \\
[\![p \lhd \tau]\!](a, \mathit{tid}) &\triangleq [\![\tau]\!].\mathrm{ghost}(a, \mathit{tid}) * ([\![\tau]\!].\mathrm{phys}(a, \mathit{tid}) = [\![p]\!]) \\
[\![p \lhd^{\dagger\kappa} \tau]\!] &\triangleq ([\dagger\kappa] \Rrightarrow\!\!\mathop{\Large *}_{\mathcal{N}_{lft}} [\![p \lhd \tau]\!])
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathbf{E}]\!] &\triangleq \mathop{\Large *}_{e\in\mathbf{E}} [\![e]\!] \\
[\![\kappa \sqsubseteq_e \kappa']\!] &\triangleq [\![\kappa]\!] \sqsubseteq [\![\kappa']\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathbf{L}]\!] &\triangleq \mathop{\Large *}_{l\in\mathbf{L}} [\![l]\!] \\
[\![\kappa \sqsubseteq_l \overline{\kappa}]\!] &\triangleq \exists \kappa'.\ ([\![\kappa]\!] = \kappa' \sqcap (\sqcap [\![\overline{\kappa}]\!])) * (\square([\kappa'] \mathrel{\Rrightarrow\!\!\!\!\!\!\mathop{\Large *}_{\mathcal{N}_{lft}}} [\dagger\kappa']))
\end{aligned}
$$

$$
[\![\mathbf{E}; \mathbf{L} \vdash \kappa \text{ alive}]\!] \triangleq [\![\mathbf{E}]\!] \vdash ([\![\mathbf{L}]\!] \twoheadrightarrow_{\mathcal{N}_{lft}} [\![\kappa]\!])
$$

$$
[\![\mathbf{I}]\!] \triangleq \text{(See Figure 6.16)}
$$

$$
\begin{aligned}
[\![\mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow_f \tau_2]\!] &\triangleq [\![\mathbf{L}]\!] \mathrel{-\!\!*} \square([\![\mathbf{E}]\!] \mathrel{-\!\!*} \tau_1 \sqsubseteq_f^{\mathsf{ty}} \tau_2) \\
&\text{where} \\
[\![\tau_1 \sqsubseteq_f^{\mathsf{ty}} \tau_2]\!] &\triangleq (\square \forall a, \mathit{tid}.\ ([\![\tau_1]\!].\mathrm{ghost}(a, \mathit{tid}) \Rightarrow [\![\tau_2]\!].\mathrm{ghost}(f(a), \mathit{tid})) \wedge \\
&\qquad (\square \forall a, \mathit{tid}.\ ([\![\tau_1]\!].\mathrm{phys}(a, \mathit{tid}) = [\![\tau_2]\!].\mathrm{phys}(f(a), \mathit{tid}))) \wedge \\
&\qquad (\square \forall a, \mathit{tid}, G.\ (G \twoheadrightarrow_{\mathcal{N}_{lft}} [\![\tau_1]\!].\mathrm{ghost}(a, \mathit{tid})) \Rightarrow \\
&\qquad\quad (G \twoheadrightarrow_{\mathcal{N}_{lft}} [\![\tau_2]\!].\mathrm{ghost}(f(a), \mathit{tid})))
\end{aligned}
$$

Figure 6.12: **Semantic models of the contexts.**

Semantic models of types

$$
\begin{array}{ll}
\lfloor \mathbf{int} \rfloor & \triangleq \mathbb{Z} \\
[\![\mathbf{int}]\!].\mathrm{ghost}(i, \mathit{tid}) & \triangleq \mathsf{True} \\
[\![\mathbf{int}]\!].\mathrm{phys}(i, \mathit{tid}) & \triangleq [i] \\
\\
\lfloor \mathbf{bool} \rfloor & \triangleq \mathbb{B} \\
[\![\mathbf{bool}]\!].\mathrm{ghost}(b, \mathit{tid}) & \triangleq \mathsf{True} \\
[\![\mathbf{bool}]\!].\mathrm{phys}(b, \mathit{tid}) & \triangleq [b] \\
\\
\lfloor \lightning_n \rfloor & \triangleq \overline{\mathit{FancyValue}} \\
[\![\lightning_n]\!].\mathrm{ghost}(\overline{v}, \mathit{tid}) & \triangleq \mathsf{True} \\
[\![\lightning_n]\!].\mathrm{phys}(\overline{v}, \mathit{tid}) & \triangleq \overline{v} \\
\\
\lfloor \mathbf{Tracked}\ \tau \rfloor & \triangleq \lfloor \tau \rfloor \\
[\![\mathbf{Tracked}\ \tau]\!].\mathrm{ghost}(a, \mathit{tid}) & \triangleq [\![\tau]\!].\mathrm{ghost}(a, \mathit{tid}) \\
[\![\mathbf{Tracked}\ \tau]\!].\mathrm{phys} & \triangleq [] \\
\\
\lfloor \mathbf{own}_n\ \tau \rfloor & \triangleq \mathit{Loc} \times \lfloor \tau \rfloor \\
[\![\mathbf{own}_n\ \tau]\!].\mathrm{ghost}((\ell, a), \mathit{tid}) & \triangleq (\ell \hookrightarrow [\![\tau]\!].\mathrm{phys}(a, \mathit{tid})) * \mathsf{dealloc}_n(\ell) * \triangleright [\![\tau]\!].\mathrm{ghost}(a, \mathit{tid}) \\
[\![\mathbf{own}_n\ \tau]\!].\mathrm{phys}((\ell, a), \mathit{tid}) & \triangleq [\ell] \\
\\
\lfloor \&^{\kappa}_{\mathrm{shr}} \tau \rfloor & \triangleq \mathit{Loc} \times \mathit{CellChain}_n \times \lfloor \tau \rfloor \\
[\![\&^{\kappa}_{\mathrm{shr}} \tau]\!].\mathrm{ghost}((\ell, \overline{\delta}, a), \mathit{tid}) & \triangleq ([\kappa] \twoheadrightarrow_{\mathcal{N}_{\mathit{lft}}} (\ell \overset{(\overline{\delta})}{\hookrightarrow} [\![\tau]\!].\mathrm{phys}(a, \mathit{tid}))) \\
 & \quad * ([\kappa] \twoheadrightarrow_{\mathcal{N}_{\mathit{lft}}} [\![\tau]\!].\mathrm{ghost}(a, \mathit{tid})) \\
[\![\&^{\kappa}_{\mathrm{shr}} \tau]\!].\mathrm{phys}((\ell, \overline{\delta}, a), \mathit{tid}) & \triangleq [\ell]
\end{array}
$$

Figure 6.13: **Semantic model of types (Part I).**

Semantic models of types (II)

$$
\begin{array}{ll}
\lfloor \mathbf{PPtr} \rfloor & \triangleq \mathit{Loc} \\
[\![\mathbf{PPtr}]\!].\mathrm{ghost}(\ell, \mathit{tid}) & \triangleq \mathsf{True} \\
[\![\mathbf{PPtr}]\!].\mathrm{phys}(\ell, \mathit{tid}) & \triangleq [\ell] \\
 & \\
\lfloor \mathbf{PointsTo}_n\ \tau \rfloor & \triangleq \mathit{Loc} \times \lfloor \tau \rfloor \\
[\![\mathbf{PointsTo}_n\ \tau]\!].\mathrm{ghost}((\ell, a), \mathit{tid}) & \triangleq (\ell \hookrightarrow [\![\tau]\!].\mathrm{phys}(a, \mathit{tid})) * \triangleright [\![\tau]\!].\mathrm{ghost}(a, \mathit{tid}) \\
[\![\mathbf{PointsTo}_n\ \tau]\!].\mathrm{phys}(a, \mathit{tid}) & \triangleq [] \\
 & \\
\lfloor \mathbf{Dealloc}_n \tau \rfloor & \triangleq \mathit{Loc} \\
[\![\mathbf{Dealloc}_n \tau]\!].\mathrm{ghost}(\ell, \mathit{tid}) & \triangleq \mathsf{dealloc}_n(\ell) \\
[\![\mathbf{Dealloc}_n \tau]\!].\mathrm{phys}(a, \mathit{tid}) & \triangleq [] \\
 & \\
\lfloor \mathbf{PCell}_n\ \tau \rfloor & \triangleq \mathit{CellId}_n \\
[\![\mathbf{PCell}_n\ \tau]\!].\mathrm{ghost}(\overline{\gamma}, \mathit{tid}) & \triangleq \mathsf{True} \\
[\![\mathbf{PCell}_n\ \tau]\!].\mathrm{phys}(\overline{\gamma}, \mathit{tid}) & \triangleq \overline{\gamma} \\
 & \\
\lfloor \mathbf{Cell::PointsTo}_n\ \tau \rfloor & \triangleq \mathit{CellId}_n \times \lfloor \tau \rfloor \\
[\![\mathbf{Cell::PointsTo}_n\ \tau]\!].\mathrm{ghost}((\overline{\gamma}, a), \mathit{tid}) & \triangleq (\overline{\gamma} \hookrightarrow [\![\tau]\!].\mathrm{phys}(a, \mathit{tid})) * \triangleright [\![\tau]\!].\mathrm{ghost}(a, \mathit{tid}) \\
[\![\mathbf{Cell::PointsTo}_n\ \tau]\!].\mathrm{phys}(a, \mathit{tid}) & \triangleq [] \\
 & \\
\lfloor \mathbf{Resource}(\mathit{RA}) \rfloor & \triangleq \mathit{Name} \times M \\
[\![\mathbf{Resource}(\mathit{RA})]\!].\mathrm{ghost}((\gamma, m), \mathit{tid}) & \triangleq \boxed{m}^{\gamma} \\
[\![\mathbf{Resource}(\mathit{RA})]\!].\mathrm{phys}(a, \mathit{tid}) & \triangleq [] \\
 & \\
\lfloor \mathbf{StorageResource}(\tau, \mathit{SP}) \rfloor & \triangleq \mathit{Name} \times P \\
[\![\mathbf{StorageResource}(\tau, \mathit{SP})]\!].\mathrm{ghost}((\gamma, p), \mathit{tid}) & \triangleq \langle p \rangle^{\gamma} * (\gamma \in \mathcal{N}_{\mathit{lft}}) \\
 & \qquad * \mathrm{sto}\left(\gamma, \lambda t.\ \textstyle\ast_{(k,v)\in t} [\![\tau]\!].\mathrm{ghost}(v, \mathit{tid})\right) \\
[\![\mathbf{StorageResource}(\tau, \mathit{SP})]\!].\mathrm{phys}(a, \mathit{tid}) & \triangleq [] \\
\end{array}
$$

Figure 6.14: **Semantic model of types (Part II).**

To prove LLftL-Begin, we can open the (6.11) (using Guard-Open) and apply (6.3) for some fresh k, then let $\kappa = \{k\}$. To kill the lifetime, we open the invariant and use (6.7).

Many rules follow immediately: LLftL-Not-Own-End, LLftL-Incl-Isect, LLftL-Tok-Unit, LLftL-End-Unit, LLftL-Dead-Persistent.

LLftL-Tok-Inter follows from RA-And, and LLftL-Incl-Glb follows from that together with Guard-And.

Proving LLftL-BorrowShared To do this we're going to put P in a "cancellable (Leaf) invariant" that requires the dead lifetime token to cancel it. First, suppose we have a basic *cancellation* resource:

$$\mathsf{True} \Rrightarrow \exists j.\ \mathsf{Cancel}\ j$$
$$\mathsf{Cancel}\ j * \mathsf{Cancel}\ j \vdash \mathsf{False}$$

Using Guard-Forever, exchange P for:

$$\exists j.\ \mathsf{Cancel}\ j * \left(\mathsf{True} \twoheadrightarrow_{\mathcal{N}_{lft}} \triangleright (P \lor ([\dagger\kappa] * \mathsf{Cancel}\ j))\right)$$

By Guard-Or-Cancel, we can eliminate the right-hand side of the disjunction when guarded by $[\kappa]$. This gives us $[\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} \triangleright P$.

Furthermore, we have:

$$\mathsf{Cancel}\ j \vdash [\dagger\kappa] \Rrightarrow\!\!\!\!\ast_{\mathcal{N}_{lft}} \triangleright P$$

By simply opening the $\twoheadrightarrow$-invariant and exchanging the cancel token and the dead lifetime token to get $\triangleright P$ back.

Proving LLftL-Incl-Dead-Implies-Dead From $[\dagger\kappa']$ and $[\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} [\kappa']$, we get $[\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} \mathsf{False}$.

Now, we take (6.11) and case on whether $\kappa \subseteq A$:

$$\mathsf{True} \twoheadrightarrow_{\mathcal{N}_{lft}} \left(\left(\exists A, D.\ (\kappa \subseteq A) * \mathsf{LtState}(A, D) * \ast_{a \in A} \mathsf{Alive}(a)\right) \lor \left(\exists A, D.\ (\kappa \not\subseteq A) * \mathsf{LtState}(A, D) * \ast_{a \in A} \mathsf{Alive}(a)\right)\right)$$

By Guard-Or-Cancel-G we can cross out the left disjunct, which contains $[\kappa]$ in the conjunct of Alive tokens. This leaves us with the case that $\kappa \not\subseteq A$, so we can open it (Guard-Open) and obtain $[\dagger\kappa]$.

6.4.3 Semantic model of the type-spec judgment

Now we can provide a semantic interpretation of the type-spec:

$$\begin{aligned}
&[\![\mathbf{E}; \mathbf{L} \mid \mathbf{I}; \mathbf{T} \vdash\ I\ \dashv\ r.\ \mathbf{I}'; \mathbf{T}' \rightsquigarrow \Phi]\!] \triangleq \\
&\quad \forall \Psi, \mathit{tid}.\ \{\exists \overline{a}, \overline{h}.\ \Phi\Psi(\overline{a}, (\overline{h})) * [\![\mathbf{L}]\!] * [\![\mathbf{T}]\!](\overline{a}, \mathit{tid}) * [\![\mathbf{I}]\!](\overline{h}, \mathit{tid})\} \\
&\qquad\quad I \\
&\qquad\quad \{r.\ \exists \overline{b}, \overline{j}.\ \Psi(\overline{b}, (\overline{j})) * [\![\mathbf{L}']\!] * [\![\mathbf{T}']\!](\overline{b}, \mathit{tid}) * [\![\mathbf{I}]\!](\mathit{tid}, \overline{j}, \mathit{tid})\}
\end{aligned}$$

This is based on a simplified version of RustHornBelt's type-spec. (Theirs is much more complicated because they handle mutable reference and prophecy variables.) Of course, we made some additional modifications to account for the invariant context, still highlighted in blue; we will discuss those below.

As we can see, this definition in turn relies on the semantic interpretations of the contexts: $[\![\mathbf{T}]\!]$, $[\![\mathbf{L}]\!]$, and so on. These interpretations are provided in Figure 6.12. The idea is actually pretty simple when you break it down:

- We want to prove the soundness of a typing judgment that allows us to transform the typing context from before an instruction to after.
- We define interpretations of these contexts as Iris resources.
- We show a Hoare tuple that the instruction lets us exchange the resources of the before-context to the resources of the after-context.

The semantic interpretation of the type context **T** is, of course, defined by the semantic interpretations of its constituent types. So let us finally talk about these.

6.4.4 Semantic models of types

In RustHornBelt, the semantic model of a type τ contains a function $[\![\tau]\!].\text{own} : \lfloor\tau\rfloor \times \mathit{ThreadId} \times \overline{\mathit{Value}} \to \mathit{iProp}$, that is, it is an Iris proposition asserting the validity of a value (as a sequence of memory words) of type τ on a given thread and with a given representation value. RustBelt and RustHornBelt also have a second function, the $[\![\cdot]\!].\text{shr}$ predicate, making it possible to specify what it means to "share" a type on a *per-type basis*. However, for λ_{Verus}, we will be using the Leaf ↠ operator to handle shared types uniformly. Therefore, in our semantic model, we will **not be using a shr predicate**. Thus far, this is probably our sharpest departure from the proof architecture of RustBelt.

There is one more thing we do slightly differently. We split up the **own** predicate into two: **ghost** and **phys**.

$$[\![\tau]\!].\text{ghost} : \lfloor\tau\rfloor \times \mathit{ThreadId} \to \mathit{iProp}$$
$$[\![\tau]\!].\text{phys} : \lfloor\tau\rfloor \times \mathit{ThreadId} \to \overline{\mathit{FancyValue}}$$

I will explain what *FancyValue* is soon, but for now, just think of it as *Value*, the sort stored in any physical memory word.

The point of the split is that we can separate the "ghost" content of a type—all the owned resources that are associated with it—from the description of the physical representation. That way we can have ghost values with the associated ghost content but no physical memory words, or real physical values that have both aspects.

Let us handle one slight bureaucratic detail. For technical reasons, the physical value needs to be constrained to a single value (or "fancy value" actually); that is why we represent it as the output of a function rather than passing it as an argument to **own**. However, the $\lfloor\tau\rfloor$ values (Figure 6.3) often do not have enough information to constrain the physical value. To fix this, one solution would be to introduce another argument; however, this would produce a lot of clutter. Instead, we simply fold extra information into $\lfloor\tau\rfloor$. This is why the definitions of $\lfloor\tau\rfloor$

in Figure 6.13 and Figure 6.14 differ than those we originally gave in Figure 6.3. This results in more precise type-specs, but it is otherwise immaterial.

Now, one might protest that constraining the physical memory value is problematic for handling interior mutability types like **PCell**, where the interior data is very much *not* constrained, not even when under a shared reference. We will come back to this point soon.

6.4.5 Proofs for PPtr and PointsTo

Hopefully, the semantic interpretations of **PointsTo** and **Dealloc** look straightforward. The **PointsTo** interpretation has the points-to, ℓ pointing to the desired physical value, and it also has the other "ghost" resources associated with τ. The **Dealloc** interpretation has the $\text{dealloc}_n(\ell)$ resource, the permission to perform a deallocation at ℓ for a range of n memory words.

With these definitions, it should hopefully look fairly straightforward to prove the type-specs for **PPtr**-related operations that we saw in Figure 6.5.

Allow me to walk through the proof for **PPtr_borrow**, which deals with shared references. In the precondition, we have:

$$[\![\text{pt} \lhd \&^{\kappa}_{\text{trk}}\textbf{PointsTo}_n\ \tau]\!]$$

Expanding all this, we end up in the definition of $[\![\&^{\kappa}_{\text{shr}}\textbf{PointsTo}_n\ \tau]\!].\text{ghost}$. You can ignore all the stuff about "cell chains" for the moment, as we just need the right half:

$$[\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} [\![\textbf{PointsTo}_n\ \tau]\!].\text{ghost}((\ell, v), \mathit{tid})$$

Continue expanding:

$$[\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} ((\ell \hookrightarrow [\![\tau]\!].\text{phys}(v, \mathit{tid})) * [\![\tau]\!].\text{ghost}(v, \mathit{tid}))$$

We can split that up (Guard-Split):

$$\big([\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} (\ell \hookrightarrow [\![\tau]\!].\text{phys}(v, \mathit{tid}))\big) * \big([\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} [\![\tau]\!].\text{ghost}(v, \mathit{tid})\big)$$

And that gives us the desired $[\![\&^{\kappa}_{\text{shr}}\tau]\!].\text{ghost}(\cdot)$. Of course, **PPtr_borrow** returns the necessary pointer, so we have a physical reference, as needed.

6.4.6 Interior mutability

By inspection of the definition of $[\![\&^{\kappa}_{\text{shr}}\tau]\!].\text{ghost}$, something seems to pose a problem for interior mutability. The value on the right-hand side of the $\hookrightarrow$ is fixed! Furthermore, I found this to be technically necessary; I originally tried to put an existential into the $\twoheadrightarrow$ operator so that the value could vary, but this ran into technical difficulties. How, then, can we ever implement interior mutability, which allows mutation through a shared reference?

The trick we use here is to use an enhanced notion of the points-to connective, $\hookrightarrow$. Specifically, we enhance it with an explicit notion of *cells*. We introduce new variants on the points-to, including $\ell \hookrightarrow \mathsf{CellId}(\gamma)$, which means "location ℓ points to a cell c" with a new variant on the connective: $\ell \hookrightarrow \mathsf{CellId}(\gamma)$. We can also say $\mathsf{CellId}(\gamma) \hookrightarrow v$, which means "the cell c has interior value v."

We can combine these connectives:

$$(\ell \hookrightarrow \mathsf{CellId}(\gamma)) * (\mathsf{CellId}(\gamma) \hookrightarrow v) \dashv\vdash (\ell \overset{(\gamma)}{\hookrightarrow} v)$$

Naturally, $\ell \overset{(\gamma)}{\hookrightarrow} v$ behaves like $\ell \hookrightarrow v$ as far as reading or writing to ℓ goes. For technical reasons, we do need to track the intermediate cells when we combine, so we cannot simply replace it with $\ell \hookrightarrow v$. Thus, we annotate the points-to with a sequence of cell IDs, denoted $\overline{\delta}$, that we call the *cell chain*.[6] The key laws about this points-to resource algebra are shown in Figure 6.15.

Note that if we want to have a cell spanning multiple machine words, we need multiple cell IDs. Let CellId_n be the sort of n-vectors of cell IDs and $\mathit{CellChain}_n$ be the sort of n-vectors of cell chains.

Let us make a few observations:

- If we have $\&^{\kappa}_{\mathrm{shr}}\mathbf{PCell}_n$ and ownership of $\mathbf{Cell::PointsTo}_n\ \tau$ then that means we have some read-only $\ell \overset{(\overline{\delta})}{\hookrightarrow} \mathsf{CellId}(\gamma)$ and a mutable $\mathsf{CellId}(\gamma) \overset{(\overline{\delta}')}{\hookrightarrow} v$. This is sufficient to write to ℓ and thus update v, since this does not require us to mutate the former.
- If we have $\&^{\kappa}_{\mathrm{shr}}\mathbf{PCell}_n$ and $\&^{\kappa}_{\mathrm{shr}}\mathbf{Cell::PointsTo}_n\ \tau$, then we have read-only $\ell \overset{(\overline{\delta})}{\hookrightarrow} \mathsf{CellId}(\gamma)$ and read-only $\mathsf{CellId}(\gamma) \overset{(\overline{\delta}')}{\hookrightarrow} v$. Thus with Cell-And we can combine these into read-only $\mathsf{CellId}(\gamma) \overset{(\overline{\delta}\gamma\overline{\delta}')}{\hookrightarrow} v$ and hence get $\&^{\kappa}_{\mathrm{shr}}\tau$.
- How can we *move* a cell? Any type needs to be moveable by a straight memcpy, and that includes moving a $\mathbf{PCell}_n$ even when we don't have the $\mathbf{Cell::PointsTo}_n\ \tau$. To do this, we use the rule Cell-Concretize:

  $$\ell \overset{(\overline{\delta})}{\hookrightarrow} \mathsf{CellId}(\gamma) \Rrightarrow \exists v.\ \ell \overset{(\overline{\delta})}{\hookrightarrow} v * \mathsf{UntetheredCell}(\gamma, v)$$

  In other words, we have the ability to get the concrete byte data that makes up the cell and obtain a token ($\mathsf{UntetheredCell}(\gamma, v)$) that lets us put the cell somewhere else with the same byte data (Cell-Renew):

  $$\ell \overset{(\overline{\delta})}{\hookrightarrow} v * \mathsf{UntetheredCell}(\gamma, v) \Rrightarrow \ell \overset{(\overline{\delta})}{\hookrightarrow} \mathsf{CellId}(\gamma)$$

6.4.7 Proofs for ghost resources

Most of these follow pretty immediately from the semantic definition of $\mathbf{Resource}(RA)$ and the basic rules for RA ghost state (4.2).

- **RA_Alloc** follows from $\mathcal{V}(m) \vdash \exists\gamma.\ \boxed{m}^{\gamma}$ (RA-Unit).
- **RA_Join** follows from $\boxed{g}^{\gamma} * \boxed{h}^{\gamma} \vdash \boxed{g \cdot h}^{\gamma}$ (RA-Sep).
- **RA_Split** follows from $\boxed{f \cdot h}^{\gamma} \vdash \boxed{f}^{\gamma} * \boxed{h}^{\gamma}$ (also RA-Sep).

[6] I would have liked to remove this detail entirely, but I could not come up with an RA that had all the necessary laws that did not involve tracking cell chains. One might hope that we could sweep it away with an existential, i.e., define something like $\ell \hookrightarrow v \triangleq \exists\overline{\delta}.\ \ell \overset{(\delta)}{\hookrightarrow} v$. However, this runs into the same problem that the existential does not play nicely with the $\twoheadrightarrow$ operator. We need to pull it all the way to the top.

PointsTo/Cell algebra

$(\gamma \in \mathit{CellId}, \overline{\delta} \in \mathit{CellChain}, \ell \in \mathit{Loc}, v \in \mathit{Value})$

Propositions: $\ell \overset{(\overline{\delta})}{\hookrightarrow} v \quad \ell \overset{(\overline{\delta})}{\hookrightarrow} \mathsf{CellId}(\gamma) \quad \mathsf{CellId}(\gamma) \overset{(\overline{\delta})}{\hookrightarrow} v \quad \mathsf{CellId}(\gamma) \overset{(\overline{\delta})}{\hookrightarrow} \mathsf{CellId}(\gamma') \quad \mathsf{UntetheredCell}(\gamma, v)$

$$(a \overset{(\overline{\delta})}{\hookrightarrow} \mathsf{CellId}(\gamma)) * (\mathsf{CellId}(\gamma) \overset{(\overline{\delta}')}{\hookrightarrow} b) \dashv\vdash (a \overset{(\overline{\delta}\gamma\overline{\delta}')}{\hookrightarrow} b)$$

Cell-And
$(a \overset{(\overline{\delta})}{\hookrightarrow} \mathsf{CellId}(\gamma)) \wedge (\mathsf{CellId}(\gamma) \overset{(\overline{\delta}')}{\hookrightarrow} b) \vdash (a \overset{(\overline{\delta}\gamma\overline{\delta}')}{\hookrightarrow} b)$

Cell-New
$(a \overset{(\overline{\delta})}{\hookrightarrow} b) \Rrightarrow \exists\gamma.\ (a \overset{(\overline{\delta}\gamma)}{\hookrightarrow} b)$

Cell-Concretize
$(\ell \overset{(\overline{\delta})}{\hookrightarrow} \mathsf{CellId}(\gamma)) \Rrightarrow \exists v.\ (\ell \overset{(\overline{\delta})}{\hookrightarrow} v) * \mathsf{UntetheredCell}(\gamma, v)$

Cell-Renew
$(\ell \overset{(\overline{\delta})}{\hookrightarrow} v) * \mathsf{UntetheredCell}(\gamma, v) \Rrightarrow (\ell \overset{(\overline{\delta})}{\hookrightarrow} \mathsf{CellId}(\gamma))$

Hoare rules

PtC-Heap-Write
$\{\ell \overset{(\overline{\delta})}{\hookrightarrow} v\}\ \ell \leftarrow v'\ \{\ell \overset{(\overline{\delta})}{\hookrightarrow} v'\}$

PtC-Heap-Write-Cell
$[\ell \overset{(\overline{\delta})}{\hookrightarrow} \mathsf{CellId}(\gamma)]\{\mathsf{CellId}(\gamma) \overset{(\overline{\delta'})}{\hookrightarrow} v\}\ \ell \leftarrow v'\ \{\mathsf{CellId}(\gamma) \overset{(\overline{\delta'})}{\hookrightarrow} v'\}$

PtC-Read-Shared
$[\ell \overset{(\overline{\delta})}{\hookrightarrow} v]\ \{\}\ !\ell\ \{r.\ v = r\}$

Figure 6.15: **PointsTo propositions enhanced with cell IDs.** *These can be derived by a resource algebra. Recall the bracket notation from §4.5.1.*

- **RA_Unit** follows from True $\vdash \boxed{\epsilon}^{\gamma}$.

The ones involving shared ghost state are a little more unusual, but they still follow from standard RA rules and the elementary Leaf rules (4.7).

- For **RA_WeakenShared**, we need to show $[\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} \boxed{g}^{\gamma} \vdash [\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} \boxed{h}^{\gamma}$. To get this, we first have $\boxed{g}^{\gamma} \twoheadrightarrow \boxed{h}^{\gamma}$ from Guard-Split. Then we apply Guard-Trans.
- For **RA_JoinShared**, we need $([\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} \boxed{g}^{\gamma}) * ([\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} \boxed{h}^{\gamma}) \vdash ([\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} \boxed{f}^{\gamma})$ for the given constraint on g, h, and f. This follows from RA-And and Guard-And.
- For **RA_Validate**, we need to use the lifetime context. It suffices to show,

  $$[\![\mathbf{L}]\!] * \boxed{g}^{\gamma} * ([\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} \boxed{h}^{\gamma}) \Rrightarrow_{\mathcal{N}_{lft}} [\![\mathbf{L}]\!] * \boxed{g}^{\gamma} * \mathcal{V}(g \cdot h)$$

  Using the condition that $\mathbf{E}; \mathbf{L} \vdash \kappa$ alive, we get $([\![\mathbf{L}]\!] \twoheadrightarrow_{\mathcal{N}_{lft}} [\kappa])$; thus we have $([\![\mathbf{L}]\!] \twoheadrightarrow_{\mathcal{N}_{lft}} \boxed{h}^{\gamma})$ by Guard-Trans. Then by Guard-Upd, what we want to show reduces to:

  $$\boxed{h}^{\gamma} * \boxed{g}^{\gamma} \Rrightarrow \mathcal{V}(g \cdot h) * \boxed{h}^{\gamma} * \boxed{g}^{\gamma}$$

  This of course follows from RA-Valid.
- For **RA_Update**, This is similar to the above. We start with RA-Update-Nondeterministic to get:

  $$\boxed{g}^{\gamma} * \boxed{h}^{\gamma} \Rrightarrow (\exists b.\ (b \in B) * \boxed{b}^{\gamma}) * \boxed{h}^{\gamma}$$

  Then with $([\![\mathbf{L}]\!] \twoheadrightarrow_{\mathcal{N}_{lft}} \boxed{h}^{\gamma})$, we apply Guard-Upd to get:

  $$\boxed{g}^{\gamma} * [\![\mathbf{L}]\!] \Rrightarrow_{\mathcal{N}_{lft}} (\exists b.\ (b \in B) * \boxed{b}^{\gamma}) * [\![\mathbf{L}]\!]$$

What about storage protocols?

- For **SP_Guard**, we have a shared reference as an input; hence, we have:

  $$[\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} \langle x \rangle^{\gamma}$$

  By using the precondition and applying SP-Guard, we also have,

  $$\langle x \rangle^{\gamma} \twoheadrightarrow_{\mathcal{N}_{lft}} F([k \mapsto s])$$

  where

  $$F = \lambda t.\ \text{✳}_{(k,v) \in t} [\![\tau]\!].\text{ghost}(v, \mathit{tid})$$

  So that's just:

  $$\langle x \rangle^{\gamma} \twoheadrightarrow_{\mathcal{N}_{lft}} [\![\tau]\!].\text{ghost}(s, \mathit{tid})$$

  Applying Guard-Trans gives us:

  $$[\kappa] \twoheadrightarrow_{\mathcal{N}_{lft}} [\![\tau]\!].\text{ghost}(s, \mathit{tid})$$

  And from there we get the $\&^{\kappa}_{\text{trk}}\tau$.

- For exchange, we apply SP-Exchange-Guarded (or its variant for handling nondeterminism).

What's going on with the masks here is actually fairly subtle:

- We need $\gamma \in \mathcal{N}_{lft}$ since we want the $\twoheadrightarrow$ proposition from SP-Guard to be used for a shared reference.
- Since $\gamma \in \mathcal{N}_{lft}$, we need to use SP-Exchange-Guarded, which we devised specifically to allow this situation.

Without SP-Exchange-Guarded, it would be impossible for a storage protocol to guard its own ghost state. This might seem like a niche scenario; however, it would be very difficult to rule out the possibility if we also want to support recursive types; for as long as we support recursive types, it is easy for the user to write a storage protocol that guards itself.

6.4.8 Semantic interpretations for atomic and local invariants

Iris has a concept called *non-atomic invariants*, which we need to define the semantic models for our **LocalInvariant** types and **AtomicInvariant** types. (Confusingly, we will be using non-atomic invariants for our **AtomicInvariant** type.)

Non-atomic invariants use a special token $[\mathsf{na} : \mathcal{E} : \mathit{tid}]$ to track which invariants are open, thus serving a similar role as the masks on view-shifts do for "standard" invariants. The rules for creating and opening invariants are shown in Figure 6.16. Note that we use a general definition for the $\boxed{\cdot}^{\iota,\mathit{tid}}_{\mathsf{na}}$ in terms of mask-changing view shifts; this is necessary to prove the sync rules for **AtomicInvariant**. The semantic interpretation of an invariant type is just an invariant constructed within this scheme. We use a collection of disjoint namespaces $\mathcal{N}_{user}(\gamma)$ for invariants with name γ.

The invariant context has a somewhat complicated definition, but it basically says:

- Every invariant name ι is either in the mask $\mathcal{E}_{user}$ or is accounted for in the context of open invariants.
- We have the $[\mathsf{na} : \mathcal{E} : \mathit{tid}]$ token for all the names that haven't been taken out by local invariants.
- For each open invariant $\iota \lhd \mathbf{InLocal}(C, \tau, I)$, we have a $\Rightarrow\!\!\!*$ that lets us close the invariant (given the appropriate ghost state satisfying the invariant).

Handling cancellable invariants Cancellable invariants are a well-known idea in Iris. Here, we want to use cancellable invariants so that we can implement **Local_Destroy**, which needs to regain ownership of the τ. In order to do this, we need to "cancel" the invariant to regain full ownership of its contents.

Usually, cancellable invariants use some kind of "fractional" token for the purpose of cancellation, but we can use Leaf instead. To implement cancellable invariants, the only ghost state we need is a single Cancel j proposition where:

$$\mathsf{Cancel}\ j * \mathsf{Cancel}\ j \vdash \mathsf{False}$$

So for example, suppose we need to open the invariant (as in the proof of **Local_Open**). We have $[\kappa] \twoheadrightarrow \lfloor \mathbf{LocalInvariant}(C, \tau, I) \rfloor$; thus we have $[\kappa] \twoheadrightarrow \mathsf{Cancel}\ j$. Then when we open the

Non-atomic invariants

$$\textsc{NaTokenInit}\quad \mathsf{True} \Rrightarrow \exists \mathit{tid}.\ [\mathsf{na} : \top : \mathit{tid}]$$

$$\textsc{NaTokenAdd}\quad \frac{\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset}{[\mathsf{na} : \mathcal{E}_1 : \mathit{tid}] * [\mathsf{na} : \mathcal{E}_2 : \mathit{tid}] \dashv\vdash [\mathsf{na} : \mathcal{E}_1 \cup \mathcal{E}_2 : \mathit{tid}]}$$

$$\textsc{NaInvInit}\quad \frac{\mathcal{N}\ \text{infinite}}{\triangleright P \Rrightarrow \boxed{P}^{\mathcal{N},\mathit{tid}}_{\mathsf{na}}}$$

$$\textsc{NaInvOpen}\quad \boxed{P}^{\mathcal{N},\mathit{tid}}_{\mathsf{na}} \dashv\vdash \Box \forall \mathcal{E}.\ (\mathcal{N} \subseteq \mathcal{E}) * [\mathsf{na} : \mathcal{E} : \mathit{tid}] \Rrightarrow (\triangleright P) * [\mathsf{na} : \mathcal{E} \backslash \mathcal{N} : \mathit{tid}] * ((\triangleright P) * [\mathsf{na} : \mathcal{E} \backslash \mathcal{N} : \mathit{tid}] \Rrightarrow\!\!\!\!* [\mathsf{na} : \mathcal{E} : \mathit{tid}])$$

Atomic invariants

$$[\mathsf{at} : \mathcal{E}] \triangleq [\mathsf{na} : \mathcal{E} : \mathit{FakeDummyThreadId}]$$
$$\boxed{P}^{\mathcal{N}}_{\mathsf{at}} \triangleq \boxed{P}^{\mathcal{N},\mathit{FakeDummyThreadId}}_{\mathsf{na}}$$

Semantic model of the invariant context

$$\llbracket \mathbf{I} \rrbracket\ :\ \lfloor \mathbf{I} \rfloor \times \mathit{ThreadId} \to \mathit{Prop}$$

$$\begin{aligned}\llbracket \mathbf{I} \rrbracket((\overline{h}, \mathcal{E}_{\mathit{user}}), \mathit{tid}) \triangleq\ & \Big(\text{*}_{(\iota,i)\in(\overline{h},\mathbf{I})} \llbracket i \rrbracket(\iota, \mathit{tid})\Big) \\ & * [\mathsf{na} : \bigcup\nolimits_{\iota \in L} \mathcal{N}_{\mathit{user}}(\iota) : \mathit{tid}] \\ & * \Big(A \neq \emptyset \mathrel{-\!\!*} \Big(\mathsf{GlobalLockCounter}(|A|) * [\mathsf{at} : \bigcup\nolimits_{\iota \in A} \mathcal{N}_{\mathit{user}}(\iota)]\Big)\Big) \\ & * (\mathcal{E}_{\mathit{user}} \cup A \cup L) = \top)\end{aligned}$$

(where L the set of local invariant ι in the context)

(where A the set of atomic invariant ι in the context)

$$\llbracket \iota \lhd \mathbf{InAtomic}(C, \tau, I) \rrbracket(\iota, \mathit{tid}) \triangleq (\exists a.\ I(c, a) * \llbracket \tau \rrbracket.\mathsf{ghost}(a, \mathit{tid})) \Rrightarrow\!\!\!\!* [\mathsf{na} : \mathcal{N}_{\mathit{user}}(\llbracket \iota \rrbracket) : \mathit{tid}]$$

$$\llbracket \iota \lhd \mathbf{InLocal}(C, \tau, I) \rrbracket(\iota, \mathit{tid}) \triangleq (\exists a.\ I(c, a) * \llbracket \tau \rrbracket.\mathsf{ghost}(a, \mathit{tid})) \Rrightarrow\!\!\!\!* [\mathsf{at} : \mathcal{N}_{\mathit{user}}(\llbracket \iota \rrbracket)]$$

Semantic models of the invariant types

$$\lfloor \mathbf{LocalInvariant}(C, \tau, I) \rfloor \triangleq \mathit{Name} \times C$$

$$\llbracket \mathbf{LocalInvariant}(C, \tau, I) \rrbracket.\mathsf{ghost}((\gamma, c), \mathit{tid}) \triangleq \exists j.\ \boxed{(\exists a.\ I(c, a) * \llbracket \tau \rrbracket.\mathsf{ghost}(a, \mathit{tid})) \vee \mathsf{Cancel}\ j}^{\mathcal{N}_{\mathit{user}}(\gamma),\mathit{tid}}_{\mathsf{na}} * \mathsf{Cancel}\ j$$

$$\llbracket \mathbf{LocalInvariant}(C, \tau, I) \rrbracket.\mathsf{phys} \triangleq []$$

$$\lfloor \mathbf{AtomicInvariant}(C, \tau, I) \rfloor \triangleq \mathit{Name} \times C$$

$$\llbracket \mathbf{AtomicInvariant}(C, \tau, I) \rrbracket.\mathsf{ghost}((\gamma, c), \mathit{tid}) \triangleq \exists j.\ \boxed{(\exists a.\ I(c, a) * \llbracket \tau \rrbracket.\mathsf{ghost}(a, \mathit{tid})) \vee \mathsf{Cancel}\ j}^{\mathcal{N}_{\mathit{user}}(\gamma)}_{\mathsf{at}} * \mathsf{Cancel}\ j$$

$$\llbracket \mathbf{AtomicInvariant}(C, \tau, I) \rrbracket.\mathsf{phys} \triangleq []$$

Figure 6.16: **Semantic models for the invariant context and invariant types.**

invariant, we can rule out the right disjunct, letting us access the part of the invariant we care about.

Handling atomic invariants While local invariants use a different invariant pool $[\mathsf{na} : \mathcal{E} : \mathit{tid}]$ per-thread, the atomic invariants use just one pool, $[\mathsf{at} : \mathcal{E}]$, not tied to a particular thread ID. We can use some invariants to tie this to the global reentrant atomic lock. We can arrange it so that, upon the first call to `acquire_reentrant_global_atomic_lock`, we get $[\mathsf{at} : \top]$, allowing us to open any atomic invariant.

6.4.9 Recursive types and the later modality

Once again, we come to the problem that to handle recursive types, we need to be able to construct $[\![\mu T.\ \tau]\!]$, even though this is defined in terms of itself. Luckily, Iris provides an answer to this. In Iris, a recursive definition of $[\![\tau]\!]$ in terms of itself is well-formed as long as each recursive occurrence of $[\![\tau]\!]$ is "behind a later ($\triangleright$)."[7]

Also recall that λ_{Verus} has a restriction on recursive types: a recursive type $\mu T.\ \tau$ is well-formed only if all appearances of T in τ occur behind a pointer type. Therefore, in order to ensure that $[\![\mu T.\ \tau]\!]$ is well-formed, we just have to ensure that the semantic model of every pointer type includes a $\triangleright$. Conveniently, being behind a $\twoheadrightarrow^{\triangleright n}$ (where $n \geq 1$) counts for this purpose. (Formally, we say that $G \twoheadrightarrow^{\triangleright n} P$ is *contractive* in P.)

Until now, I have not said much about the later-counts on the $\twoheadrightarrow^{\triangleright n}$ guards. In fact, these counts actually "build up" over time. For example, consider the application of SP-Guard in the derivation of the **SP_Guard** type-spec. From SP-Guard we can get a guard with one later: $G \twoheadrightarrow^{\triangleright 1} P$. Now suppose we composed that with an existing borrow: $[\kappa] \twoheadrightarrow^{\triangleright n} P$. Applying Guard-Trans, we end up with $[\kappa] \twoheadrightarrow^{\triangleright n+1} P$. Also recall that to *apply* a $\twoheadrightarrow^{\triangleright n}$ (e.g., in Guard-Later-Open or SP-Exchange-Guarded-Nondeterministic-Later) we need to take n later steps.

Fortunately, this is not actually problematic. The count n in a $\twoheadrightarrow^{\triangleright n}$ guards proposition can be bounded by the program step counter. Furthermore, (like RustHornBelt) we can use *flexible step-indexing* using *time receipts* [57] in order to advance by the requisite number of step-indices in a given program step. (RustHornBelt used flexible step-indexing for reasons that seem closely related to mutable references. It is interesting that we need it here for what seems to be an unrelated reason.)

6.4.10 Atomic and non-atomic memory

Proving a rule like PtC-Read-Shared is nontrivial for the semantics of non-atomic memory. (This has nothing to do with the cell chains—that is completely orthogonal.) The issue has to do with the way that λ_{Rust} and λ_{Verus} define non-atomic semantics. Since a non-atomic read or write is two consecutive steps, a non-atomic read is not *truly* read-only. It actually changes the heap state in order to implement the dynamic semantics that trigger a 'stuck state' upon detection of a data race. The extended Leaf paper [27] details how to handle this.

[7] This is often called *guarded recursion*, which may be confusing because I have been using "guard" extensively for a much different purpose. To make matters more confusing, a Leaf $\twoheadrightarrow$ can act as a later guard for this purpose, as I will explain in about three sentences. The clash of terminology is, of course, entirely my fault.

6.5 On termination of ghost code

As explained in §6.1, λ_{Verus} does not handle the termination of ghost code, which is necessary for full soundness of the actual Verus implementation, which erases all ghost code. Here, I'll explain a bit about how Verus ensures termination of ghost code and the considerations that arise, though I will not provide a rigorous formal argument.

6.5.1 A paradox to watch out for

In the implementation, Verus ensures termination of spec code by detecting recursive definitions and ensuring that they all use well-formed *decreases-measures* (i.e., values that must decrease on each recursive call). However, there are some pitfalls to watch out for with this approach, especially when it comes to our *invariant* types (`AtomicInvariant` and `LocalInvariant`).

This paradox can be explained by looking at *Landin's Knot* [41], a means of encoding recursion through a combination of higher-order functions and mutable state. It is difficult to write concisely in Rust; here it is in a functional ML-style:

```
let r = ref (fun () -> ()) in
r := (fun () -> (! r) ()) ;
(! r) ()
```

The key idea is that we: **(i)** create a reference for storing a function (initialized with a dummy function that doesn't matter), **(ii)** create a function that reads the reference and calls the resulting function, and **(iii)** put this new function into the reference, then call it. The code has no explicit recursion in it—no definition dependent on itself—so this nonterminating code is not caught by ordinary recursive definition checks.

Now, for our purposes, invariants can play the "role" of a mutable reference. Therefore, in combination with higher-order functions, naively implemented invariants would permit nonterminating ghost code. It takes some additional work to actually leverage the nontermination into a proof of `false`, but this can be done with the help of RA ghost state. In fact, Upamanyu Sharma has shown an explicit construction of how the paradox would be expressible in Verus if certain higher-order features were supported in ghost code [67]. Such features include "ghost-mode **dyn**" (existential types, effectively) and "ghost-mode closures."

It should be noted that this all has an analogue in CSL. The Iris logic also has to work hard to avoid a similar paradox with its invariants [71]. This is the reason Iris has the later modality ($\triangleright$); the Inv-Open rule without the later modality would be unsound. However, Verus has no analogue of the later-modality, so we need to resolve this a different way.

6.5.2 Resolving the paradox

Currently, this is not a problem in Verus because Verus does not support ghost-mode closures or ghost-mode **dyn**. We have no need for any of these features in any of the case studies, so I could end the story here.

However, it is still worth considering how we could support such features in the future. To prepare for this eventuality, we have implemented a system based on Iris's *later credits* [69]. The

idea is that we require a "credit" in order to open an invariant; as in Iris, Verus's credits can be obtained only during executable (non-ghost) machine steps.

Now, we can argue that ghost code terminates as follows (and this is admittedly somewhat handwavey): Between any two executable steps, there can only be a finite number of credits available. Thus, we can only open a finite number of invariants. Furthermore, any ghost code which never opens an invariant must eventually terminate [43]. Ergo, the ghost code between these two executable instructions terminates.

Since Verus has yet to implement the relevant higher-order features, it remains to be seen how well this system works out in practice.

6.6 The Verus TCB

The *trusted computing base (TCB)* of a verification framework is the part of the framework that needs to be correct for the result to be completely sound. All else equal, smaller TCBs are better because it means less surface area for a mistake to invalidate the verification methodology.

Verus, unfortunately, has a fairly large TCB, which includes:

- The rustc compiler, including type-checking, lifetime-checking, and codegen.
- Verus, including its verification generation and the automated theorem provers it uses: Z3 plus specialized solvers (§3.3.3).
- The Verus primitives (Table 3.2), including both their specifications and their implementations in terms of Rust unsafe code.
- The verification condition generation of VerusSync.

Having the compiler in the TCB is hardly unusual, though it is worth noting that some Rust verification tools *are* able to remove lifetime-checking from the TCB (see §10.3). By contrast, Verus's reliance on rustc's lifetime-checking is a key pillar of its design, and it is a factor in Verus's efficiency.

λ_{Verus} provides abstract evidence that the Verus primitives and core ideas of verification condition generation are sound, though λ_{Verus} has a number of limitations (§6.1), and it is not formally connected to Verus in any way.

For VerusSync, we likewise have an on-paper sketch of its soundness (§5.5), but again, it is not mechanized.

6.7 Recap

λ_{Verus} helps us gain confidence in the foundation of Verus by establishing a formal connection between the Verus primitives and the CSL concepts they are based on. The type-spec system of λ_{Verus} simultaneously explains why Verus's specification-checking is entirely orthogonal to lifetime-checking, while also explaining why the storage protocol's cornerstone guarding operation manifests as the bounded-lifetime type signature, `(&'a T) -> (&' S)`. In fact, working out this formalization helped identify a soundness issue in Verus's implementation of invariant contexts (§6.3.3).

Despite basing the language and syntax closely on λ_{Rust}, and using the same high-level approach to logical type soundness, our actual semantic models differed sharply. One of the principles of RustBelt was the idea that every type decides what it means to be shared. In λ_{Verus}, by contrast, we handle sharing in a uniform way using Leaf's $\twoheadrightarrow$ operator. I hope that this effort gives a new perspective on Rust's sharing types.

It should be noted that our goals, from the outset, were slightly different from RustBelt. RustBelt was intentionally designed as open-ended and extensible, and while I would hardly say extensibility is a *non-goal* of λ_{Verus}, it is certainly de-emphasized. From the beginning, we had a fixed set of primitive types and operations we wanted to handle.

Of course, one significant limitation of λ_{Verus} is the lack of mutable references. To bridge this gap would require two things: First, we would need to extend the Leaf Lifetime Logic to include an equivalent of RustBelt's "full borrow" system. Second, we would need to implement RustHornBelt's *parametric prophecies*. I am cautiously optimistic in these things, as I believe they are largely orthogonal to the innovations made here, which nearly all concern shared borrows. Nonetheless, this remains future work.

Chapter 7

Specifications, Refinement, and the Global State Machine

The reader may notice we have yet to say anything about a solution to Challenge SpC-1. We will finally approach the problem in this chapter.

7.1 Specifications and refinement

There are a few ways to write a specification for a program. The most common way to do it the Hoare-style via pre-conditions and post-conditions, which is ideal for its simplicity and composability. However, it is not the only way.

Another way is to consider possible *execution traces* of the program, recording externally observable events, and then specifying what traces are allowed. This enables types of reasoning that might be difficult or impossible to do within the program logic, though what this means could vary per application.

A second advantage is that these execution traces can often be expressed as state machines, and we can then prove things about them via an established technique called *state machine refinement*.

In this chapter, I will describe *the GSM method*, which is a particular way of establishing a trace-based specification for a program that uses ghost tokens. We first introduced this method in our paper on the IronSync framework in Linear Dafny [28], though of course we did not have VerusSync then; there, we described everything in terms of monoids.

The GSM method will be used in **two** of our four case studies: Node Replication (§9.2) and SplinterCache (§9.1). For most of this chapter, it will be more helpful to keep SplinterCache and Challenge SpC-1 in mind, as the SplinterCache's application of the GSM method sits closer to the motivations we will encounter in this chapter.

7.2 IronFleet and VeriBetrKV

As a stepping stone, I will start by explaining an approach taken by IronFleet. (It may be apparent that IronFleet is the namesake of IronSync.)

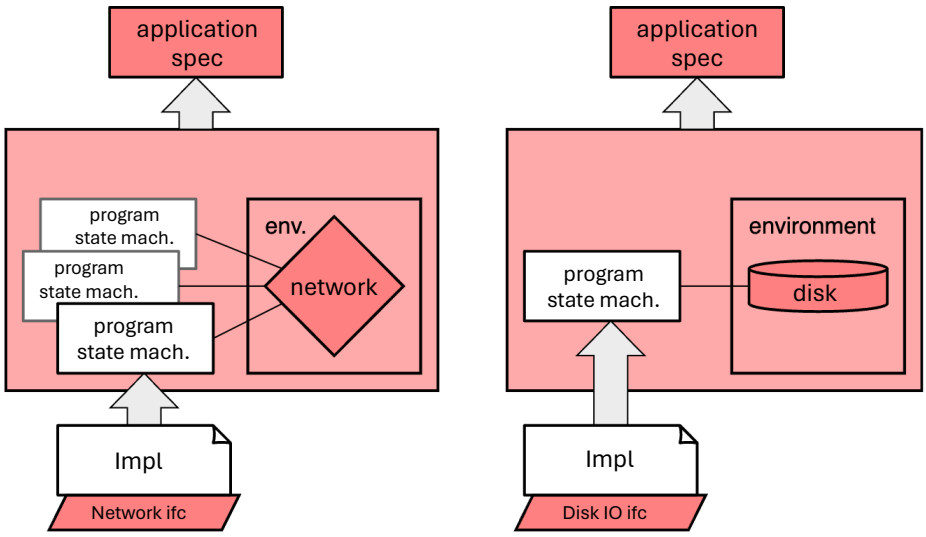


Figure 7.1: **Comparison of IronFleet and VeriBetrKV.** *Figure is from the VeriBetrKV paper [25]. Pink indicates a trusted component.*

IronFleet [29] is a framework for verifying distributed systems. They verified *IronRSL*, a Paxos-based replicated state machine, and *IronKV*, a sharded key-value store. Relatedly, VeriBetrKV [25] (my own work) is a verified, crash-safe key-value store that uses a method heavily inspired by IronFleet. Now, none of the case studies in this thesis involve distributed systems, but the SplinterCache *is* a storage system, so VeriBetrKV's observations about the commonalities in their approach are relevant to us.

Roughly, the roadmap is:

- Establish an abstraction of your program called the *program abstraction*, and prove that your program obeys the abstraction. This abstraction captures all externally-visible events, such as user requests and IO actions.
- Create a *system abstraction*: a model of the the system as a whole, defined generically over the program abstraction, describing how the program interacts with its environment.
- Prove that the system abstraction meets some desired specification.

In IronFleet, which was concerned with distributed systems, the "environment" described n copies of the program abstraction interacting over a network, and the network model contained all the assumptions about packet loss, packet duplication, and so forth. In VeriBetrKV, the environment includes a storage disk and the queue of outstanding IO operations in between the program and the disk. The environment definition captures crashes and disk corruption. We need a similar environment to address the SplinterCache case study, which also uses a storage disk. See Figure 7.1.

Observe a few things about this picture. It primarily has two *trusted* components. The specification, of course, is trusted, in that the specification needs to be meaningful in order to

learn anything meaningful about the system behavior. The system abstractions's description of the environment is also trusted: that is, anything we learn about the behavior of the system is conditional on the environment model correctly describing the environment. However, neither the *program abstraction*, nor of course the *program* itself, is trusted. The program abstraction is just an intermediate step, and it does not appear in the end-to-end theorem which relates the program to the environment and the spec.

Now, to explain this picture in a bit more detail, I will break the picture into two halves.

(1) The "top half," where we reason about the program abstraction and its interactions with the environment.

(2) The "bottom half," where we make the argument that a given program faithfully implements the abstraction.

For the top half, we largely follow what IronFleet and VeriBetrKV did; for the bottom half, we will introduce the new GSM method.

7.3 The top half: the system abstraction and refinement

First, we abstract the behavior of a program as a sequence of *events*; we consider two types of events, *user events* ($u \in \mathit{UserEvents}$) and *IO events* ($\mathit{io} \in \mathit{IOEvents}$). Let,

$$\mathit{ProgramEvents} = \mathit{UserEvents} \cup \mathit{IOEvents}$$

The user events represent any interaction visible to the user (e.g., "the user initiates operation x" or "the user receives a response"), and the IO events represent interactions with the environment (e.g., "the program sends a message" or "the program receives a message").

Whereas the behavior of the program can be described as a *trace* of user events and IO events, the behavior of the system as a whole will be described as a trace of user events only (as IO events are considered to be internal to the system).

In the most general case, we can describe "the allowed behaviors of the program" as a predicate on traces of *ProgramEvents*, and likewise "the allowed behaviors of the system" as a predicate on traces of *UserEvents*. An *environment model* is anything which takes such a description about the program and returns a description of the system.

Most of the time, though, we would prefer not to work directly with traces. In practice, it is usually more intuitive to work with state machines. Specifically, we usually work with *labeled state transition systems*. For a set of labels *Labels*, define a labeled state transition system to be a set of states $\mathcal{S}$, a specification of valid initial states $\mathit{Init} : \mathcal{S} \to \text{Bool}$, and a transition relation $\tau : \mathcal{S} \times \mathcal{S} \times \mathit{Labels} \to \text{Bool}$.

A labeled state transition system very naturally gives rise to traces of label events. We'll consider label sets $\mathit{ProgramLabels} = \mathit{ProgramEvents} \cup \{\mathit{None}\}$ and $\mathit{SystemLabels} = \mathit{SystemEvents} \cup \{\mathit{None}\}$. The *None* transition allows the system to make a transition without supplying an event for the trace.

Example: An environment with a storage disk Here we define an environment with a storage disk. Let $(\mathcal{S}_{\mathit{prog}}, \mathit{Init}_{\mathit{prog}}, \tau_{\mathit{prog}})$ be a labeled transition system with labels $\mathit{ProgramLabels} =$

$\{\mathit{None}\} \cup \mathit{UserEvents} \cup \mathit{IOEvents}$ where

$$\mathit{IOEvents} \;:=\; \mathit{ReadReq}(d) \mid \mathit{ReadResponse}(d, x) \mid \mathit{WriteReq}(d, x) \mid \mathit{WriteAck}(d)$$

Here, $d : \mathit{DiskIdx}$ is the index of a disk page and $x : \mathit{Data}$ is some page-sized data. This is meant to model a program that interacts with asynchronous disk IO actions; the program can make a read request and later get back a read response with the data, or it can make a write request and later get an acknowledgment.

We can define a labeled transition system $(\mathcal{S}_{sys}, \mathit{Init}_{sys}, \tau_{sys})$ with labels $\{\mathit{None}\} \cup \mathit{UserEvents}$ as follows. First define:

$$\mathcal{S}_{sys} \triangleq \mathcal{S}_{prog} \times (\mathit{Multiset\ IOEvents}) \times (\mathit{DiskIdx} \to \mathit{Data})$$

The state here consists of: **(i)** the program state, **(ii)** a collection of in-progress IO operations, and **(iii)** a disk state, as a mapping of disk indices to disk pages. Then the initializations and transitions:

$$\begin{aligned}
\mathit{Init}_{sys}((s_{prog}, q, r)) &\triangleq \mathit{Init}_{prog}(s_{prog}) \wedge (q = \emptyset) \\
\tau_{sys}(s_{sys}, s'_{sys}, \ell) &\triangleq \mathit{ProgramStep}(s_{sys}, s'_{sys}, \ell) \\
&\qquad \vee \mathit{ProgramIOAction}(s_{sys}, s'_{sys}, \ell) \\
&\qquad \vee \mathit{DiskIOAction}(s_{sys}, s'_{sys}, \ell) \\
&\qquad \vee \mathit{CrashAndReboot}(s_{sys}, s'_{sys}, \ell)
\end{aligned}$$

where

$$\begin{aligned}
&\mathit{ProgramStep}((s_{prog}, q, r), (s'_{prog}, q', r'), \ell) \triangleq \\
&\qquad \tau_{prog}(s_{prog}, s'_{prog}, \ell) \wedge q = q' \wedge r = r' \wedge \\
&\qquad (\ell = \mathit{None} \vee \ell \in \mathit{UserEvents}) \\
&\mathit{ProgramIOAction}((s_{prog}, q, r), (s'_{prog}, q', r'), \ell) \triangleq \\
&\qquad \ell = \mathit{None} \wedge \exists \ell'.\ \tau_{prog}(s_{prog}, s'_{prog}, \ell') \wedge (\ell' \in \mathit{IOEvents}) \wedge (r = r') \wedge (\\
&\qquad\qquad (\exists d.\ \ell' = \mathit{ReadRequest(d)} \wedge q' = q \cup \{\mathit{ReadRequest}(d)\}) \\
&\qquad\qquad \vee (\exists d, x.\ \ell' = \mathit{WriteRequest(d,\ x)} \wedge q' = q \cup \{\mathit{WriteRequest}(d, x)\}) \\
&\qquad\qquad \vee (\exists d.\ \ell' = \mathit{ReadResponse(d)} \wedge q' = q \backslash \{\mathit{ReadRequest}(d)\}) \\
&\qquad\qquad \vee (\exists d, x.\ \ell' = \mathit{WriteAck(d,\ x)} \wedge q' = q \backslash \{\mathit{WriteRequest}(d, x)\}) \\
&\qquad) \\
&\mathit{DiskIOAction}((s_{prog}, q, r), (s'_{prog}, q', r'), \ell) \triangleq \\
&\qquad s_{prog} = s'_{prog} \wedge (\ell = \mathit{None}) \wedge (\\
&\qquad (\exists d.\ q' = q \backslash \{\mathit{ReadRequest}(d)\} \cup \{\mathit{ReadResponse}(d, r(d))\} \wedge r' = r) \\
&\qquad \vee (\exists d, x.\ q' = q \backslash \{\mathit{WriteRequest}(d, x)\} \cup \{\mathit{WriteAck}(d)\} \wedge r' = r[d \mapsto x]) \\
&\qquad) \\
&\mathit{CrashAndReboot}((s_{prog}, q, r), (s'_{prog}, q', r'), \ell) \triangleq \\
&\qquad \mathit{Init}_{prog}(s'_{prog}) \wedge q' = \emptyset \wedge r' = r \wedge \ell = \mathit{CrashAndRestart}
\end{aligned}$$

In other words, a valid transition on the system is any of the following:

- The program performs one of its internal actions, or interactions with the user
- The program performs an IO action, either initiation an IO request or processing its response, while the system's state for the in-progress IO operations is appropriately updated. (Note that the label is an IO event from the perspective of $\mathcal{S}_{sys}$, while the label is *None* from the perspective of $\mathcal{S}_{prog}$.)
- One of the in-progress IO operations reads or modifies the disk.
- The system crashes and reboots, clearing in-flight IO operations and reinitializing the program state.

This crash model is actually a bit simplistic, not accounting for corner cases like torn-writes, for example. VeriBetrKV actually had a more complicated environment model, allowing non-checksum-preserving disk corruptions. The one presented here is closer to the one we use for the SplinterCache case study.

State Machine Refinement Another reason the labeled transition system view is appealing is that it allows *state machine refinement* reasoning.

Definition 1 (State machine refinement) *Let $(\mathcal{S}, \mathit{Init}_S, \tau_S)$ and $(\mathcal{T}, \mathit{Init}_T, \tau_T)$ be labeled transition systems over Labels. We say the former refines the latter, if there is a relation $\mathcal{R} : \mathcal{S} \times \mathcal{T} \to$ Bool such that,*

$$\forall s.\ \mathit{Init}_S(s) \Rightarrow \exists t.\ \mathit{Init}_T(t) \wedge \mathcal{R}(s, t)$$

and,

$$\forall s_1, s_2, t_1, \ell : \mathcal{S}.\ \tau_S(s_1, s_2, \ell) \wedge \mathcal{R}(s_1, t_1) \Rightarrow \exists t_2.\ \tau_T(t_1, t_2, \ell) \wedge \mathcal{R}(s_2, t_2)$$

By a straightforward inductive argument, observe that a state machine refinement implies that any valid label trace of $(\mathcal{S}, \mathit{Init}_S, \tau_S)$ is a valid label trace of $(\mathcal{T}, \mathit{Init}_T, \tau_T)$. I stated the definition using a relation $\mathcal{R}$ for generality, but in practice, a function $\mathcal{S} \to \mathcal{T}$ is usually enough.

This provides a natural way to prove a specification for our system: We simply state and prove a refinement theorem from $(\mathcal{S}_{sys}, \mathit{Init}_{sys}, \tau_{sys})$ to some specification, $(\mathcal{S}_{spec}, \mathit{Init}_{spec}, \tau_{spec})$. The picture is summarized in Figure 7.2

State machines and theorem proving How do we actually mechanize state machine composition and state machine refinement theorems? Recall that our program verifier also serves as a general-purpose proof-checking engine. Thus, we can set up all these theorems and prove them using Dafny or Verus without any issues. Since this theorem isn't tied to the underlying program logic, we could in principle use any other proof-checking engine we desire, but there isn't a compelling reason to use a different one.

7.4 The bottom half: the GSM method

For the bottom half of this picture, where we need to connect the implementation to some program abstraction, the GSM method deviates sharply from IronFleet. This section will be

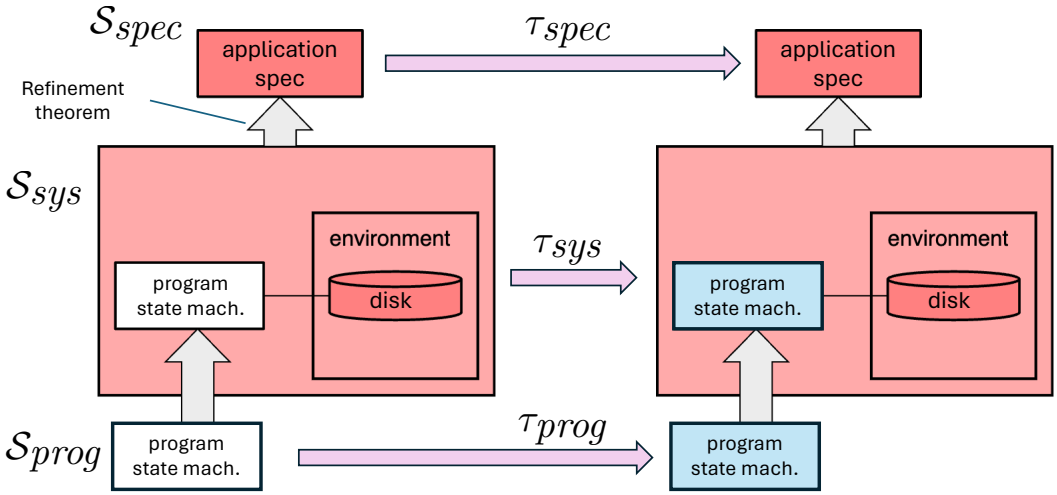


Figure 7.2: **Illustration of a refinement stack with a disk environment.** *A transition of the $\mathcal{S}_{prog}$ state machine can be lifted to a transition of $\mathcal{S}_{sys}$. $\mathcal{S}_{sys}$ is then proved to refine $\mathcal{S}_{spec}$.*

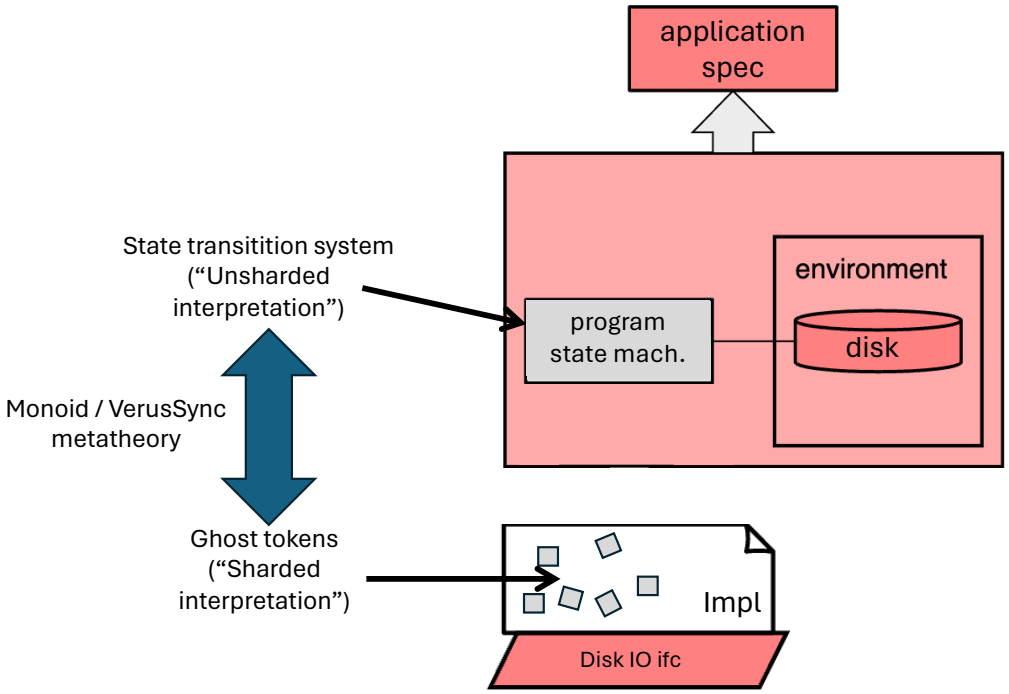


Figure 7.3: **The GSM method: VerusSync integrated into the refinement stack.**

devoted to the GSM method; for more on IronFleet and how the GSM method compares, see §10.4.

Now, the central issue is that we need to establish that a program is a faithful implementation of the program abstraction, or conversely, that the program abstraction is a faithful abstraction of the implementation. Fortunately, we already have a way to create an abstraction of a concurrent program as a transition system: We have VerusSync.

Before this chapter, all uses of VerusSync have been "inward-facing," in the sense that its main export is the token interface which is then used by some verified code as an internal proof detail. In such uses, we do not actually use the state-abstraction for anything outside of the VerusSync definition, but as a state transition system, it is actually well-positioned for a state machine refinement-based approach like the above. When used this way, we call the VerusSync instance a *Global State Machine (GSM)*. The main issue is that transitions need to be labeled with user events.

For simplicity, consider some concurrent system with a public API consisting of a single function, `do_operation(Input) -> Output`. We can describe a trace execution with two "user events": *Request*(*id*, *input*) which corresponds to the user invoking a call, and *Response*(*id*, *output*) which corresponds to its completion. The two are tied together by a "request ID" *id*.

We can create two "special transitions," one which corresponds to the injection of a request, and once which corresponds to the consumption of a response:[1]

```
fields {
    #[sharding(map)] pub requests: Map<RequestId, Request>,
    #[sharding(map)] pub responses: Map<RequestId, Response>,
    // ...
}

// Label: Request(request_id, input)
transition!{
    new_request(input: Input) {
        birds_eye let request_id = /* fresh request ID */;
        add requests += [ request_id => Request(input) ];
    }
}

// Label: Response(request_id, output)
transition!{
    complete_response(request_id: RequestId, output: Output) {
        remove responses -= [ request_id => Response(output) ];
    }
}
```

As usual, the developer also defines a handful of "normal" VerusSync transitions that get turned into ghost token operations through the usual sharding mechanism. These "normal" transitions all have label *None*; i.e., they don't correspond to user-visible actions.

However, because these two special transitions have labels that tie them to externally-observable events, we do *not* get token transition operations that can be invoked arbitrarily.

[1] VerusSync does not actually have built-in support for these special transitions; when we used the GSM method to verify NR in Verus, we manually specified the 'special' transitions and other related proof obligations externally to any VerusSync definition that defined the normal transitions. However, I thought it would be easiest to explain the idea using the transition notation already ubiquitous in this document.

How do we actually tie these special transitions to the user's invocations of the `do_operation` call? We can do so with the specification on `do_operation`:

```
fn do_operation(input: Input, Tracked(request_token): Tracked<Request>)
    -> (output: Output, Tracked(response_token): Tracked<Response>)
    requires request_token.value() == input
    ensures response_token.value() == output
        && response_token.id() == request_token.id()
```

With a specification like this, we can reason as follows:

- The invocation of `do_operation` corresponds to the injection of the `Request` token (the `new_request` transition, with label $\mathit{Request}(\mathit{id}, \mathit{input})$).
- The completion of the `do_operation` call corresponds to the consumption of the `Response` token (the `new_response` transition, with label $\mathit{Response}(\mathit{id}, \mathit{output})$).
- Other transitions are performed internally by the concurrent system, but these have label *None* and are thus ignored by the trace.

As a result, we can conclude that the GSM, that is, the labeled transition system $(\mathcal{S}_{prog}, \mathit{Init}_{prog}, \tau_{prog})$ that results from the VerusSync instance is a proper abstraction of the observable behavior of the system.

And we can do something similar to handle the *IOEvents*; we can have special transitions with *IOEvents* labels, special tokens that correspond to IO requests and responses, and a (trusted) interface for IO operations that operates on these tokens.

7.5 Limitations

One major limitation of this system means that it needs to be the last step in the proof of how a system behaves. It is not really obvious how to package up all the refinement theorems and environmental reasoning into a "ordinary" function specification that can be called from additional verified code. However, this is arguably inherent to the approach: one of the main points is to reason about interactions with external components that aren't part of the program logic in the first place.

7.6 Recap

We presented a framework for reasoning about system behavior using state machine refinement, an extension of our existing tool for reasoning via transition systems. Though transition systems are a classic tool in formal methods, the way these transition are connected to the programs via the sharding mechanism is, as far as I know, novel.

The motivation for all of this was based in the idea that we need to reason about programs and their interactions with external environments. However, we actually will only use this idea once, for the SplinterCache case study. Even if we ignore the part about the environment, the method is still a convenient way to use *state machine refinement*, which is useful in its own right. We will use this for the NR case study.

Chapter 8

Linear Dafny vs. Verus

In this chapter, I will discuss some of the technical differences between Linear Dafny and Verus. This is mostly so the reader can gain an understanding of how the methodology in this thesis has evolved on the way to its current incarnation in Verus. It is also partly in preparation for explaining our SplinterCache case study, which was done in Linear Dafny.

8.1 Monoids vs. VerusSync

Linear Dafny did not have VerusSync. Instead, Linear Dafny used something similar to the Verus Monoidal Ghost Interface. It was a *bit* different; for one thing, Linear Dafny does not have traits, so we axiomatized the interface using a module system instead. Still, the end result was similar: To use the system, the user would provide a monoid, prove it meets various definitions, and as a result they would get access to some ghost token type.

However, the very fact that we invented VerusSync is evidence enough that we found the Monoidal Ghost Interface system to be lacking in user-friendliness. One of the issues was conceptual: thinking in terms of monoid composition remained technically challenging, and as a result, we desired to have a system that positioned things like more classical transition systems, which we discussed already in Chapter 5.

Another issue with the monoidal ghost interface was the massive amount of boilerplate code involved in using the monoidal interface:

- We had to explicitly define the composition operation $(\cdot)$ and write boilerplate proof code to prove commutativity and associativity. Similarly, we had to define the unit ϵ and its well-formedness properties.
- We frequently had to reason about partiality—i.e, predicates always needed to specify "this element of the resource algebra is not the 'fail' state."
- We constantly had to reason about invariant predicates of element compositions, e.g., $\mathit{Inv}(a \cdot b)$ or even $\mathit{Inv}(a \cdot b \cdot c)$.
- Many fields of our monoid state were of the `Option<T>` type when in VerusSync they can just be `T` of the `variable` strategy.
- Packing up the frame-preserving updates into friendlier interfaces with easier-to-use

token types was laborious, requiring extensive boilerplate that dwarfed the main proofs of interest. I believe the difficulty of this step is a weakness of the ghost object style; i.e., this likely would not have been a problem in CSL. Verus does not really fix this issue, though VerusSync does side-step it.

The boilerplate reduction due to VerusSync in quantified in §9.5.3.

I also recall that the need to reason about compositions sometimes tripped up the theorem prover in ways that it VerusSync never does, though I have no concrete data about it.

8.2 References and lifetimes

Linear Dafny has an ownership type system that is inspired by Rust's, but which is slightly simpler. Linear Dafny has:

- "Linear" variables, which behave basically the same way as Rust's variables do by default; i.e., they are move-only. Unlike Rust's variables, Linear Dafny's linear variables are not dropped automatically.
- "Shared" variables, basically shared references. Shared variables could be borrowed from linear variables, though Linear Dafny lacked a full-fledged lifetime system, and instead, it had fairly restrictive scoping rules. It does, at least, support `(shared X) -> shared Y` functions, which of course were essential for the equivalent of the `Resource<P>::guard` operation.
- "In-out" parameters, similar to the restricted mutable references that Verus has.

Fortunately, the lack of lifetime variables never ended up being a problem for the Splinter-Cache or NR case studies. However, I strongly suspect it would have been a serious problem for the memory allocator case study, which (being done in Verus/Rust) currently uses non-lexical lifetimes in nontrivial ways. I think it would have been very difficult to work around this issue if the memory allocator had been done in Linear Dafny.

Furthermore, in the VeriBetrKV project [25] (one of the driving forces behind the original development of Linear Dafny), there was one data structure that was impossible to port into the linear types style because it involved a struct with a shared reference, whereas in Rust this would be easy to handle with lifetime variables:

```
1 struct X<'a> {
2     y_ref: &'a Y,
3 }
```

8.3 Atomics and Invariants

Recall that Verus supports atomic mutable cells the same way it supports non-atomic interior mutability, that is, with a physical cell type and a ghost memory permission. The only difference between the two types of memory being that the atomic instructions are considered *atomic* for the sake of atomic invariant blocks.

By contrast, Linear Dafny had a single primitive type that wrapped the physical atomic cell and the ghost state invariant all into one. The resulting type was similar to Verus's atomic-with-

ghost library that we introduced in §3.5.4. This meant that every invariant had to be tied to some physical state—though we eventually bolted on a "GhostAtomic" type in an *ad hoc* manner. Effectively this was an "Atomic" without the physical part; i.e., it was just an `AtomicInvariant`. For the most part, this didn't really matter to the user—after all, in Verus, we usually use the atomic-with-ghost library anyway, and in fact, the atomic-with-ghost library's macros were designed to resemble Linear Dafny's syntax. To the user, it was mostly all the same. However, as a *primitive*, it was much more complicated.

Linear Dafny does not have the equivalent of `LocalInvariant` at all. This wouldn't have been possible—Linear Dafny did not have anything resembling the `Send`/`Sync` marker trait system that Rust has, so it could not have enforced the critical restriction of a `LocalInvariant`, that being that it is never shared across threads.

Chapter 9

Analysis of Case Studies

In this chapter, we will cover all four case studies in depth. For each case study, we will give a technical overview for the system operation, describe the specification and Trusted Computing Base (TCB) for the verified system, describe the high-level points of the verified system, and cover the key challenges described in Chapter 2. Of course, the TCB is always implicitly taken to include all of Verus (or Linear Dafny) and its metatheory.

9.1 Case Study I: Splinter Cache

9.1.1 Specification and TCB

SplinterCache, as a component of SplinterDB, actually has a fairly involved interface. The interface is primarily based on *locks*. The client can select a disk page d and ask for a read-lock on that page; the cache will return a pointer to some cache entry containing that page and a promise that this page is read-locked. Later, the client can upgrade the lock to a write-lock.

As you might expect, our verified version has a specification for this interface in terms of ghost permissions and ghost lock tokens. However, it is a somewhat complicated spec that takes some work to interpret. Therefore, we also have a "wrapper interface" that supports simple operations like "read page," "write page," and "havoc page."[1] These operations are then specified using a GSM (§7.4). The GSM is combined with an environment model as described in that section to construct a system abstraction $(\mathcal{S}_{sys}, Init_{sys}, \tau_{sys})$. Then we show that this refines a specification $(\mathcal{S}_{spec}, Init_{spec}, \tau_{spec})$. Figure 9.1 presents a simplified version of it, not accounting for crashes.

In plain language, this simplified spec says the valid transitions of the system are:

- A request for a read, write, or havoc enters the system.
- A request is processed, thus turning into a response. For write and havoc requests, this modifies the main system state: the *dataMap*.

[1]The reason we care about a *havoc* operation is because SplinterCache provides an option for the client to get access to a page without actually loading it in from disk. This is useful if the client plans to write to the page immediately. However, this operation, on its own, effectively means overwriting the contents of the page with whatever junk data happens to be in the cache entry at that time. This is easily modeled as a havoc operation.

$$\begin{aligned}
\mathit{UserEvents} \triangleq\ & \{\mathit{ReadRequest}(\mathit{id}, d), \mathit{WriteRequest}(\mathit{id}, d, \mathit{data}),\\
& \mathit{ReadResponse}(\mathit{id}, \mathit{data}), \mathit{WriteResponse}(\mathit{id}),\\
& \mathit{HavocRequest}(\mathit{id}, d), \mathit{HavocResponse}(\mathit{id})\}
\end{aligned}$$

$$\mathcal{S}_{\mathit{spec}} \triangleq (\mathit{Multiset\ UserEvents}) \times (\mathit{DiskIdx} \to \mathit{Data})$$

$$\mathit{Init}_{\mathit{spec}}((e, \mathit{dataMap})) \triangleq e = \emptyset$$

$$\begin{aligned}
&\tau_{\mathit{spec}}((e, \mathit{dataMap}), (e', \mathit{dataMap}'), \ell) \triangleq\\
&\quad (\ell = \mathit{None} \land e = e' \land \mathit{dataMap} = \mathit{dataMap}')\\
&\quad \lor (\ell = \mathit{None} \land \exists \mathit{id}, d.\ e' = e \setminus \{\mathit{ReadRequest}(\mathit{id}, d)\} \cup \{\mathit{ReadResponse}(\mathit{id}, \mathit{dataMap}[d])\}\\
&\qquad \land \mathit{dataMap}' = \mathit{dataMap})\\
&\quad \lor (\ell = \mathit{None} \land \exists \mathit{id}, d, \mathit{data}.\ e' = e \setminus \{\mathit{WriteRequest}(\mathit{id}, d, \mathit{data})\} \cup \{\mathit{WriteResponse}(\mathit{id})\}\\
&\qquad \land \mathit{dataMap}' = \mathit{dataMap}[d \mapsto \mathit{data}])\\
&\quad \lor (\ell = \mathit{None} \land \exists \mathit{id}, d, \mathit{junkHavocData}.\ e' = e \setminus \{\mathit{HavocRequest}(\mathit{id}, d)\} \cup \{\mathit{HavocResponse}(\mathit{id})\}\\
&\qquad \land \mathit{dataMap}' = \mathit{dataMap}[d \mapsto \mathit{junkHavocData}])\\
&\quad \lor (\ell \in \{\mathit{ReadRequest}(\cdot), \mathit{WriteRequest}(\cdot), \mathit{HavocRequest}(\cdot)\} \land e' = e \cup \{\ell\} \land d' = d)\\
&\quad \lor (\ell \in \{\mathit{ReadResponse}(\cdot), \mathit{WriteResponse}(\cdot), \mathit{HavocResponse}(\cdot)\} \land e' = e \setminus \{\ell\} \land d' = d)
\end{aligned}$$

Figure 9.1: **Simplified system specification for the SplinterCache.**

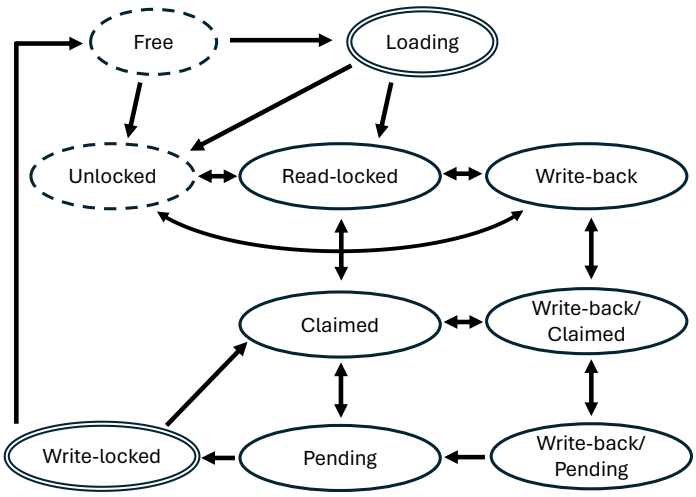


Figure 9.2: **SplinterCache internal locking system.** *This is a rough abstraction that does not include, e.g., intermediate states while checking or modifying reference counts. A dashed oval indicates an unlocked state; a solid oval indicates a state where the cache entry's backing memory is read-locked; a double oval indicates a state where the cache entry's backing memory is write-locked.*

- A response for a read, write, or havoc leaves the system.

The real one, which accounts for crashes, has two versions of the *dataMap*, a "persistent" version, and a "current" version, with the current version reverting to the persistent version on crash. The resulting guarantees on system crash are fairly weak, as generally speaking it is not a main *objective* of a cache to provide strong crash guarantees. I mostly consider this to be a "proof-of-concept" for crash-related reasoning, rather than a full-fledged study of it.

9.1.2 SplinterCache locking mechanism

Recall that that there are two levels of locking in SplinterCache. The client can request a lock on a disk page d, and internally, the cache has a locking system on each cache entry c.

The user lock has three possible states: a page d can be read-locked, write-locked, or claimed. The "claimed" state is the only one that should need any explanation. A claim is a type of read-lock with the exclusive right to upgrade to a write-lock. It is, however, not itself a write-lock, so for the most part, we treat it like a special kind of read-lock.

Now, let us turn attention to the internal lock on a cache entry c. Figure 9.2 shows an approximate state diagram of the possible states of this locking system.

Loading a page into memory To load a page, we first move from the Free to the Loading state. This write-locks the cache memory, which is necessary because we need to load data from the disk into the cache memory. After the loading is complete, we either downgrade to a normal read-lock state (Read-locked) or release the lock entirely (Unlocked). Keep in mind that the lock itself does not actually care that the *Loading* state is used for a disk load; from the lock's

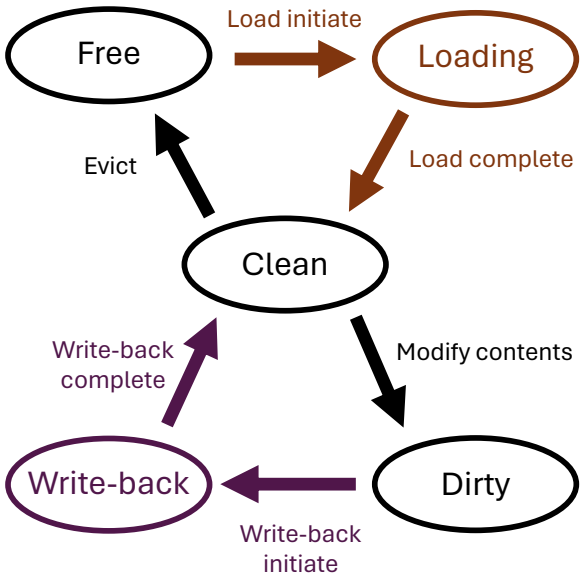


Figure 9.3: **SplinterCache cache entry lifecycle.**

perspective, it is just a kind of exclusive write-lock. It is only named *Loading* because that is how it happens to be used in practice.

Taking a read-lock for the client From Unlocked, you can move to Read-locked by incrementing a read-lock counter and checking that no exclusive lock is held.

Upgrading a read-lock to a claim This is easy to do by setting the "claim" bit on the status field. The cache just has to check that it is not already claimed or write-locked. A claim can coexist with other read-locks, including the read-lock used for write-back.

Upgrading a claim to a write-lock First, set the status field for the exclusive lock. This brings you to one of the Pending states. Wait for all existing read-locks to be released (including the write-back lock if necessary). This brings us into the Write-locked state.

Taking a read-lock for the purpose of write-back In order to write a cache entry back to disk, we have to read-lock the relevant memory for this operation to be safe. To do this, we set the write-back bit in order to move to the Write-back state. (Since the write-back bit is on the same memory word as the exclusive-lock bit and other relevant bits, we can do this with an atomic compare-and-swap.)

Proving the lock correct To prove the lock correct, we use a storage protocol similar in style to the one from the example lock earlier (§3.6.4), but with more states and fields. We

name this storage protocol CacheRwLock. In IronSync, we formally defined CacheRwLock using IronSync's version of the monoidal interface. A snippet defining the main state and the monoidal composition operator are shown in Figure 9.5. For the ease of understanding how the CacheRwLock works, though, I have also provided a version "translated" into VerusSync (Figure 9.5).

CacheRwLock has the following notable fields:

- The `storage` field, to store the relevant ghost state (memory permission for the cache entry and ghost state from the Cache system we will discuss later).
- The `flag` field, an enum with states that correspond roughly to Figure 9.2. The resulting ghost token can be related to the atomic status field in its various bit combinations.
- The `ref_counts`. Again, similar to the ref-count from §3.6.4 but this time we have multiple of them (to reduce thread contention). Anyone taking a read-lock needs to increment one of them, while anyone taking a write-lock needs to check that all of them are zero.
- The `shared_state`, `exc_state`, `loading_state`, and `writeback_state` are the states that related to all the kinds of different locks. Note that there can be multiple `shared_state` tokens at any given time, since there might be multiple readers, but there is always at most one of any other kind of state. Also note that since some of these lock kinds are multi-step processes, the state fields are enums that include variants for different points in the process. For example, the `SharedState` enum contains three possible states:
  - `Pending`, which means we incremented some refcount (tracking which one)
  - `Pending2`, which means we incremented some refcount and checked that we are not write-locked (but not yet checked that are we are done loading)
  - `Obtained`, which means we have done all necessary checks and thus have a shared read-lock.

In this system, we have two different ghost tokens that can serve as guards for the read-locks. The "normal" read-lock is done with a `shared_state` token (in the `Obtained` variant, of course) while the write-back read-lock is done with the `writeback_state` token.

Meanwhile, the exclusive write-lock and the loading write-lock states allow the user to withdraw the ghost state.

9.1.3 High-level cache properties

Being able to verify the locks, on its own, ought to be enough to prove memory safety. However, we still need to verify that the client sees consistent behavior; e.g., if they write to some disk page, and then read from it later, they should read back the same value. To do this, we use a GSM system, Cache.

Naturally, the GSM has special tokens that correspond to the *UserEvents* and *IOEvents*. More interesting are the tokens that correspond to the cache's internal state. Recall the following components of the cache's internal state:

- An array of atomics, called the *disk page map*, where each entry corresponds to a disk page d and maps that disk page to a cache entry c (or none).

```
datatype Flag =
  | Unmapped | Reading | Reading_ExcLock | Available
  | Claimed | Writeback | Writeback_Claimed | Writeback_PendingExcLock
  | PendingExcLock | ExcLock_Clean | ExcLock_Dirty

type ThreadId = nat
type StoredType = CacheHandle.Handle

// Standard flow for obtaining a 'shared' lock
datatype SharedState =
  | SharedPending(ghost t: int)
  | SharedPending2(ghost t: int)
  | SharedObtained(ghost t: int, b: StoredType)

// Standard flow for obtaining an 'exclusive' lock
datatype ExcState =
  | ExcNone
  | ExcClaim(ghost t: int, b: StoredType)
  | ExcPendingAwaitWriteback(ghost t: int, b: StoredType)
  | ExcPending(ghost t: int, ghost visited: int, clean: bool, b: StoredType)
  | ExcObtained(ghost t: int, clean: bool)

datatype WritebackState =
  | WritebackNone
  | WritebackObtained(b: StoredType)

datatype ReadState =
  | ReadNone
  | ReadPending                          // set status bit to ExcLock | Reading
  | ReadPendingCounted(ghost t: int)     // inc refcount
  | ReadObtained(ghost t: int)           // clear ExcLock bit

datatype CentralState =
  | CentralNone
  | CentralState(flag: Flag, stored_value: StoredType)

// The carrier of the monoid, M
datatype M = M(
  central: CentralState,
  ghost refCounts: map<ThreadId, nat>,
  ghost sharedState: FullMap<SharedState>,
  exc: ExcState,
  read: ReadState,
  writeback: WritebackState
) | Fail

// Definition of ·
function dot(x: M, y: M) : M {
  if
    x.M? ∧ y.M?
    ∧ !(x.central.CentralState? ∧ y.central.CentralState?)
    ∧ x.refCounts.Keys ⋔ y.refCounts.Keys
    ∧ (x.exc.ExcNone? ∨ y.exc.ExcNone?)
    ∧ (x.read.ReadNone? ∨ y.read.ReadNone?)
    ∧ (x.writeback.WritebackNone? ∨ y.writeback.WritebackNone?)
  then
    M(
      if x.central.CentralState? then x.central else y.central,
      union_map(x.refCounts, y.refCounts),
      add_fns(x.sharedState, y.sharedState),
      if !x.exc.ExcNone? then x.exc else y.exc,
      if !x.read.ReadNone? then x.read else y.read,
      if !x.writeback.WritebackNone? then x.writeback else y.writeback
    )
  else
    Fail
}
```

Figure 9.4: **Linear Dafny code for CacheRwLock.** *Snippet of the Linear Dafny code for the CacheRwLock storage protocol defining the monoidal · operator. For ease of understanding, I have translated it into "modern" VerusSync syntax in Figure 9.5.*

```
type StoredType = /* ghost state to store in lock */;
pub type BucketId = nat;

pub enum Flag { ... }

pub enum SharedState {
   Pending{bucket: BucketId},
   Pending2{bucket: BucketId},
   Obtained{bucket: BucketId, value: StoredType},
}

pub enum ExcState {
    Claim{bucket: Option<BucketId>, value: StoredType},
    PendingAwaitWriteback{bucket: Option<BucketId>, value: StoredType},
    Pending{bucket: Option<BucketId>, visited_count: BucketId,
              clean: bool, value: StoredType},
    Obtained{bucket: Option<BucketId>, clean: bool},
}

pub enum LoadingState {
    Pending,
    PendingCounted{bucket: Option<BucketId>},
    Obtained{bucket: Option<BucketId>},
}

pub struct WritebackState {
    pub value: StoredType,
}

// CacheRwLock VerusSync

fields {
    #[sharding(storage_option)] pub storage: Option<StoredType>,
    #[sharding(variable)]       pub flag: Flag,
    #[sharding(map)]            pub ref_counts: Map<BucketId, nat>,
    #[sharding(multiset)]       pub shared_state: Multiset<SharedState>,
    #[sharding(option)]         pub exc_state: Option<ExcState>,
    #[sharding(option)]         pub loading_state: Option<LoadingState>,
    #[sharding(option)]         pub writeback_state: Option<WritebackState>,
}

// Guard properties of the RwLock

property!{
    borrow_shared_obtained(ss: SharedState) {
        require let SharedState::Obtained { bucket, value } = ss;
        have shared_state >= { ss };
        guard storage >= Some(value);
    }
}

property!{
    borrow_writeback(ws: WritebackState) {
        have writeback_state >= Some(ws);
        guard storage >= Some(ws.value);
    }
}
```

Figure 9.5: **CacheRwLock, translated into VerusSync.**

Original IronSync version of Cache

```
// Monoid carrier M
datatype M = M(
  ghost disk_idx_to_cache_idx: map<nat, Option<nat>>,
  ghost entries: map<nat, Entry>,
  ghost statuses: map<nat, Status>,
  ghost write_reqs: map<nat, DiskIfc.Block>,
  ghost write_resps: set<nat>,
  ghost read_reqs: set<nat>,
  ghost read_resps: map<nat, DiskIfc.Block>,
  ghost tickets: map<RequestId, IOIfc.Input>,
  ghost stubs: map<RequestId, IOIfc.Output>,
  ghost sync_reqs: map<RequestId, set<nat>>,
  ghost havocs: map<RequestId, nat>
) | Fail

predicate InitiateLoad(s: M, s': M,
    cache_idx: nat, disk_idx: nat) {
  ∧ s.M?
  ∧ cache_idx in s.entries
  ∧ s.entries[cache_idx] == Empty
  ∧ disk_idx in s.disk_idx_to_cache_idx
  ∧ s.disk_idx_to_cache_idx[disk_idx] == None
  ∧ s' == s
    .(entries := s.entries[cache_idx := Reading(disk_idx)])
    .(disk_idx_to_cache_idx := s.disk_idx_to_cache_idx[disk_idx := Some(cache_idx)])
    .(read_reqs := s.read_reqs + {disk_idx})
}
```

Translated VerusSync version of Cache

```
Cache {
    fields {
        #[sharding(map)] pub disk_idx_to_cache_idx: Map<DiskIdx, Option<CacheIdx>>,
        #[sharding(map)] pub cache_entry: Map<CacheIdx, Entry>,
        #[sharding(map)] pub status: Map<CacheIdx, Status>,
        #[sharding(map)] pub write_reqs: Map<DiskIdx, Block>,
        #[sharding(set)] pub write_resps: Set<DiskIdx>,
        #[sharding(set)] pub read_reqs: Set<DiskIdx>,
        #[sharding(map)] pub read_resps: Map<DiskIdx, Block>,
        #[sharding(map)] pub tickets: Map<RequestId, Input>,
        #[sharding(map)] pub stubs: Map<RequestId, Output>,
        #[sharding(map)] pub sync_reqs: Map<RequestId, SyncReq>,
        #[sharding(map)] pub havoc: Map<RequestId, DiskIdx>,
    }

    transition!{
        initiate_load(cache_idx: CacheIdx, disk_idx: DiskIdx) {
            remove cache_entry -= [ cache_idx => Entry::Empty ];
            remove disk_idx_to_cache_idx -= [ disk_idx => None ];
            add cache_entry += [ cache_idx => Entry::Reading{disk_idx} ];
            add disk_idx_to_cache_idx += [ disk_idx => Some(cache_idx) ];
            add read_reqs += set {disk_idx};
        }
    }

    // ... and 10 other transitions
}
```

Figure 9.6: **"Initiate load" transition of Cache.** *Presented in its original IronSync form and via a VerusSync translation. Also see the visual in Figure 9.7*

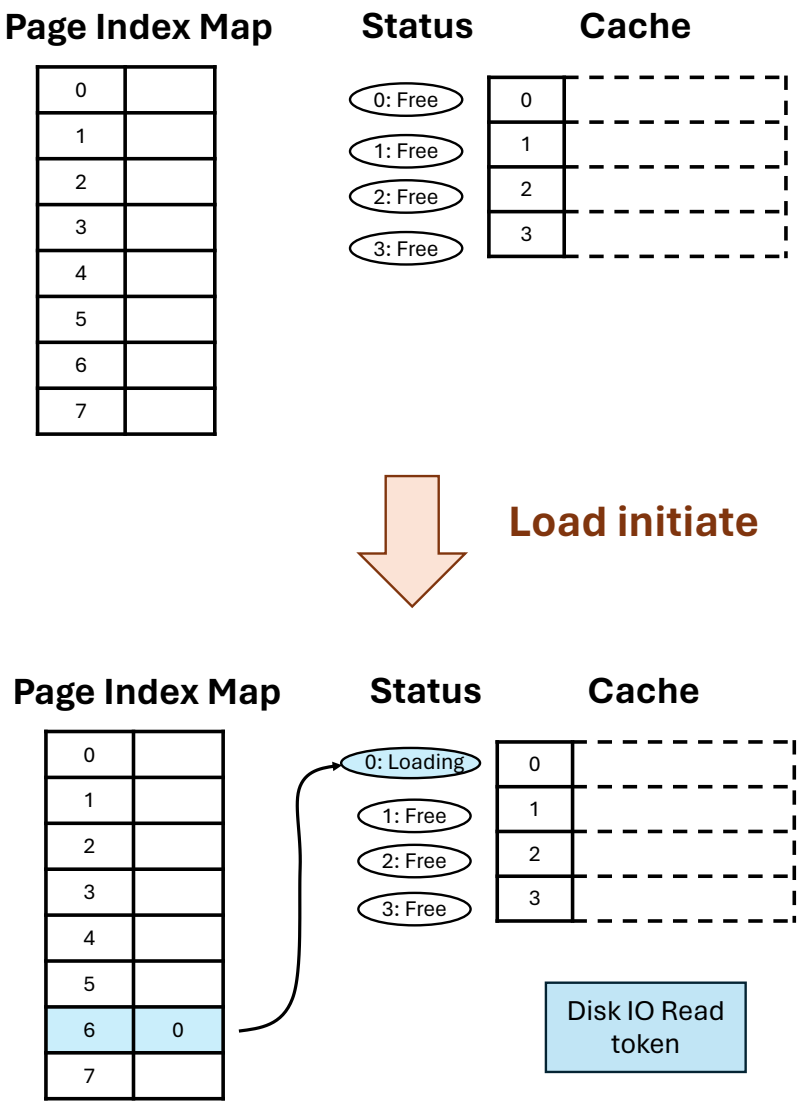


Figure 9.7: **Graphical depiction of the "Load initiate" transition in the CACHE GSM.** *We **(i)** assign disk page 6 to cache entry 0, **(ii)** set the cache entry to the Loading state, and **(iii)** create a token to perform a disk read. The cache entry remains empty.*

- Each cache entry has a *status*, a *disk address* (the reverse mapping of the above bullet point) and of course the *page data*. The status field, as discussed above, is fairly involved and serves multiple purposes. For the sake of this section, we can abstract the entry status into *five* possible states (Figure 9.3).
  - Each cache entry starts in the Free state.
  - When loading begins, it moves to the Loading state, and when loading completes, it moves to the Clean state, which means that the contents match that of the disk.
  - In order to modify the contents, it must move to the Dirty state.
  - After being written back to disk, it moves back to the Clean state.
  - When evicted, it moves from the Clean state back to the Free state.

The GSM contains ghost tokens for all of this state. Specifically:

- There is a ghost token for each entry of the disk page map, which is associated with the corresponding atomic memory location.
- There is a ghost token for each cache entry "status," which is associated with the atomic status field. (This means that the status field is associated with multiple ghost tokens, one from CacheRwLock and this one.)
- There is a ghost token for each cache entry containing the disk address and page data. This is protected by the reader-writer lock discussed above.

Figure 9.6 depicts an example transition, in both the original and the "translated" style. Figure 9.7 depicts the same transition graphically.

Once our GSM Cache is defined, we get a labeled transition system $(\mathcal{S}_{prog}, Init_{prog}, \tau_{prog})$, and thus by the environmental model, the system abstraction as well: $(\mathcal{S}_{sys}, Init_{sys}, \tau_{sys})$. Then we need to prove a refinement theorem between $(\mathcal{S}_{sys}, Init_{sys}, \tau_{sys})$ and $(\mathcal{S}_{spec}, Init_{spec}, \tau_{spec})$.

This is actually a relatively easy part of the proof, as one might guess purely from the fact that Figure 9.3 is indeed a much less intimidating figure than Figure 9.2. There are only a handful of fairly straightforward invariants we need to prove:

- The disk page map is consistent with the reverse mapping.
- The status fields are all consistent with the outstanding IO operations.
- If a cache entry is in the Clean state, then the page data matches the data on disk.

From these, all the relevant refinement theorems follow.

9.2 Case Study II: Node Replication

9.2.1 Specification and TCB

Recall that the objective of NR is to take some user-provided, sequential data structure, which we will call data structure **X**, and provide *concurrent* access to it in a highly-parallelizable way. The implementation and specification, thus, are generic over **X**, which is specified via a trait (Figure 9.8).

To summarize this trait: It first defines the operations abstractly. Operations are split into *queries* that do not mutate the data structure state, and *updates* that do. Queries are abstractly specified by `dispatch_spec` which takes as input the state of the data structure (`Self::View`) and a `ReadOperation`, and returns a `Response`. Updates are abstractly specified by `dispatch_spec_mut` which is similar, taking a `WriteOperation` and also returning a `Response`, but this time *also* returning an updated `Self::View`. The trait, of course, also has executable functions that are tied to these spec functions via postconditions.

Once again: the data structure **X** and the implementation of this trait are provided by the client, so our specification needs to be generic over types that implement this trait. The goal is linearizability, which is actually a fairly straightforward thing to specify using the state machine specification style (Figure 9.9). We call this state machine spec "Linearized(**X**)."

9.2.2 Proof Overview

Before we can explain the proofs, we need to dive deeper into the technical architecture of the system.

Is the NR system, threads are associated to logical *nodes*. At a practical level, all the threads at a given node are pinned to a single *NUMA node*. Each node maintains its own instance of **X**, which we call its *replica*. (See Figure 9.10.) The way NR handles any given operation depends on whether it is a query or an update.

- To perform a query operation, we simply query against the node's replica. In this case, the only inter-node communication required is that which is necessary to make sure the node replica is sufficiently up-to-date.
- To perform an update operation, we also need to communicate the operation to the other nodes so that they can perform the same update on their replicas. This is done via the *message buffer*.

The central algorithm for this communication process is called **execute**. Roughly speaking, it works as follows:

- One thread on a node is selected as the *executor thread*. Other threads on the node send their update operations to the executor thread. This process is called *flat-combining*.
- The executor thread inserts the update operations on the message buffer.
- The executor thread then reads all the update operations off the message buffer that it has not processed yet. It executes these operations against its node's replica. This necessarily includes all the operations added in the previous step ("node-local operations"), and it may also include operations that were placed into the buffer by other threads ("remote

```rust
pub trait Dispatch: Sized {
    // Argument to a read-only operation.
    type ReadOperation: Sized;

    // Argument to a write operation.
    type WriteOperation: Sized + Send;

    // The type on the value returned by the data structure when a
    // `ReadOperation` or a `WriteOperation` successfully executes against it.
    type Response: Sized;

    // Self is the concrete type, View is the abstraction
    type View;
    spec fn view(&self) -> Self::View;

    //// Specification of the data structure in terms of the abstract type

    spec fn init_spec() -> Self::View;
    spec fn dispatch_spec(ds: Self::View, op: Self::ReadOperation) -> Self::Response;
    spec fn dispatch_mut_spec(ds: Self::View, op: Self::WriteOperation)
        -> (Self::View, Self::Response);

    //// Executable methods

    // Initialize the data structure
    fn init() -> (res: Self)
        ensures res@ == Self::init_spec();

    // Method on the data structure that allows a read-only operation to be
    // executed against it.
    fn dispatch(&self, op: Self::ReadOperation) -> (result: Self::Response)
        ensures Self::dispatch_spec(self@, op) == result;

    // Method on the data structure that allows a write operation to be
    // executed against it.
    fn dispatch_mut(&mut self, op: Self::WriteOperation) -> (result: Self::Response)
        ensures Self::dispatch_mut_spec(old(self)@, op) == (self@, result);

    fn clone_write_op(op: &Self::WriteOperation) -> (res: Self::WriteOperation)
        ensures op == res;

    fn clone_response(op: &Self::Response) -> (res: Self::Response)
        ensures op == res;
}
```

Figure 9.8: **Trait providing a generic specification for the data structure X.**

$$
\begin{aligned}
\mathit{UserEvents} &\triangleq \{\mathit{ReadRequest}(\mathit{id}, \mathit{readOp}), \mathit{WriteRequest}(\mathit{id}, \mathit{writeOp}), \\
&\qquad \mathit{Response}(\mathit{id}, \mathit{response})\} \\
\mathcal{S}_{spec} &\triangleq (\mathit{Multiset\ UserEvents}) \times \mathit{View} \\
\mathit{Init}_{spec}((e, \mathit{dataMap})) &\triangleq e = \emptyset \land \mathit{View} = \mathit{init\_spec()} \\
\tau_{spec}((e, \mathit{view}), (e', \mathit{view}'), \ell) &\triangleq \\
&(\ell = \mathit{None} \land e = e' \land \mathit{view} = \mathit{view}') \\
&\lor (\ell = \mathit{None} \land \exists \mathit{id}, \mathit{readOp}. \\
&\quad e' = e \setminus \{\mathit{ReadRequest}(\mathit{id}, \mathit{readOp})\} \cup \{\mathit{Response}(\mathit{id}, \mathit{dispatch}(\mathit{view}, \mathit{readOp}))\} \\
&\quad \land \mathit{view}' = \mathit{view}) \\
&\lor (\ell = \mathit{None} \land \exists \mathit{id}, \mathit{writeOp}, \mathit{response}. \\
&\quad e' = e \setminus \{\mathit{WriteRequest}(\mathit{id}, \mathit{writeOp})\} \cup \{\mathit{Response}(\mathit{id}, \mathit{response})\} \\
&\quad \land (\mathit{view}', \mathit{response}) = \mathit{dispatch\_mut}(\mathit{view}, \mathit{writeOp}) \\
&\lor (\ell \in \{\mathit{ReadRequest}(\cdot), \mathit{WriteRequest}(\cdot)\} \land e' = e \cup \{\ell\} \land \mathit{view}' = \mathit{view}) \\
&\lor (\ell \in \{\mathit{Response}(\cdot)\} \land e' = e \setminus \{\ell\} \land \mathit{view}' = \mathit{view})
\end{aligned}
$$

Figure 9.9: **System specification for NR.**

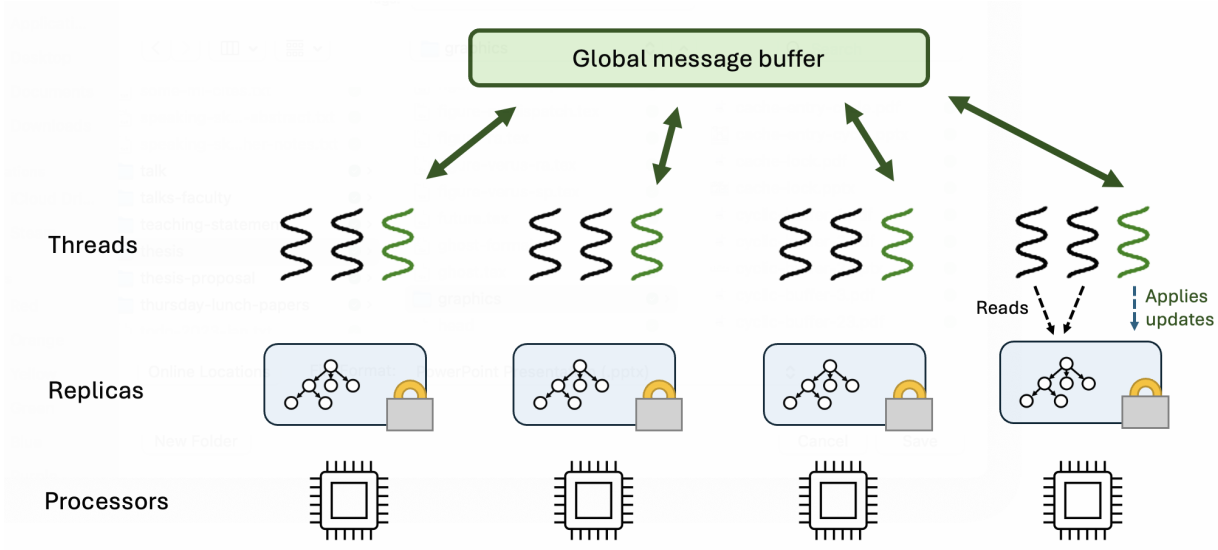


Figure 9.10: **Node Replication Architecture Overview.**

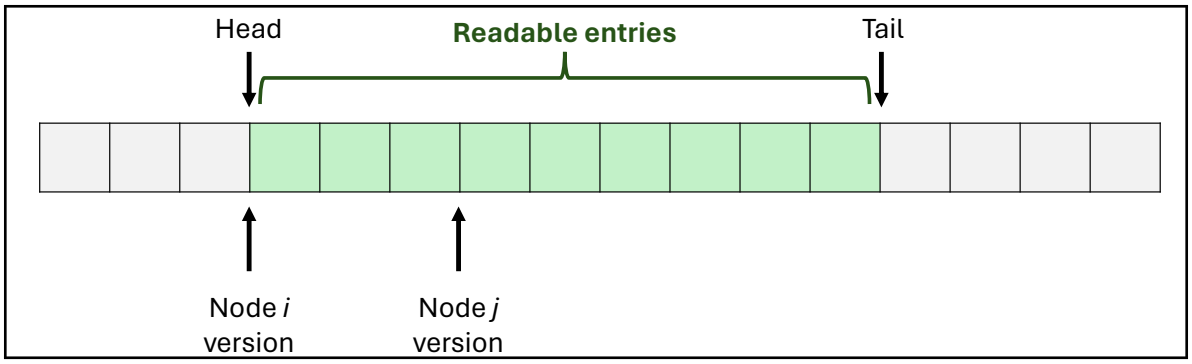


Figure 9.11: **Visual depiction of the message buffer.**

operations").

- The executor thread sends the results of the node-local operations back to the threads where the operations originated.

As we can see, the executor thread has two roles: it both *sends* a node's messages to the other threads, and it also handles messages that are *received* from other threads. Even when performing a query operation, it may be necessary to call **execute**, since it may be necessary to update the replica with remote updates. On the whole, this process ensures that all threads process the same update operations in the same order.

The Message Buffer

The message buffer is a fixed sized buffer where indices "wrap around." A variety of indices are used to maintain the state of the buffer (Figure 9.11).

- The *tail* pointer points to the next empty entry which will be used for the next message.
- For each node, there is a pointer to the end of the range that it has read from.
- It is necessary that the tail does not get sufficiently far ahead that we accidentally wrap around and overwrite some messages that have not been read by all nodes yet. To do this, we keep a *head* pointer that marks a spot known to be safe for the tail to reach.

When an executor thread is ready to insert messages onto the buffer, it first increments the tail pointer (atomically) in order to reserve some entries. Then it writes the messages into those entries and marks them as *alive* (indicating that they are okay for reading). This process is illustrated in Figure 9.12.

With all this established, we now understand the "lifecycle" of a given memory cell in the buffer. A cell is first reserved, written to, and then marked active; from then on, it is readable by any node until all of the node version pointers have passed it. Then it can be reserved again. This illustrates the read-shared / writable-exclusive dichotomy that storage protocols can handle.

The first major component of the proof, then, is a storage protocol we call CyclicBuffer (Figure 9.13). CyclicBuffer allows the storage of one `cell::PointsTo` per entry. A thread can withdraw an entry's `cell::PointsTo` by incrementing the *tail* pointer. It deposits the `cell::PointsTo` back into the system when it marks the entry as "alive." Finally, it is possible

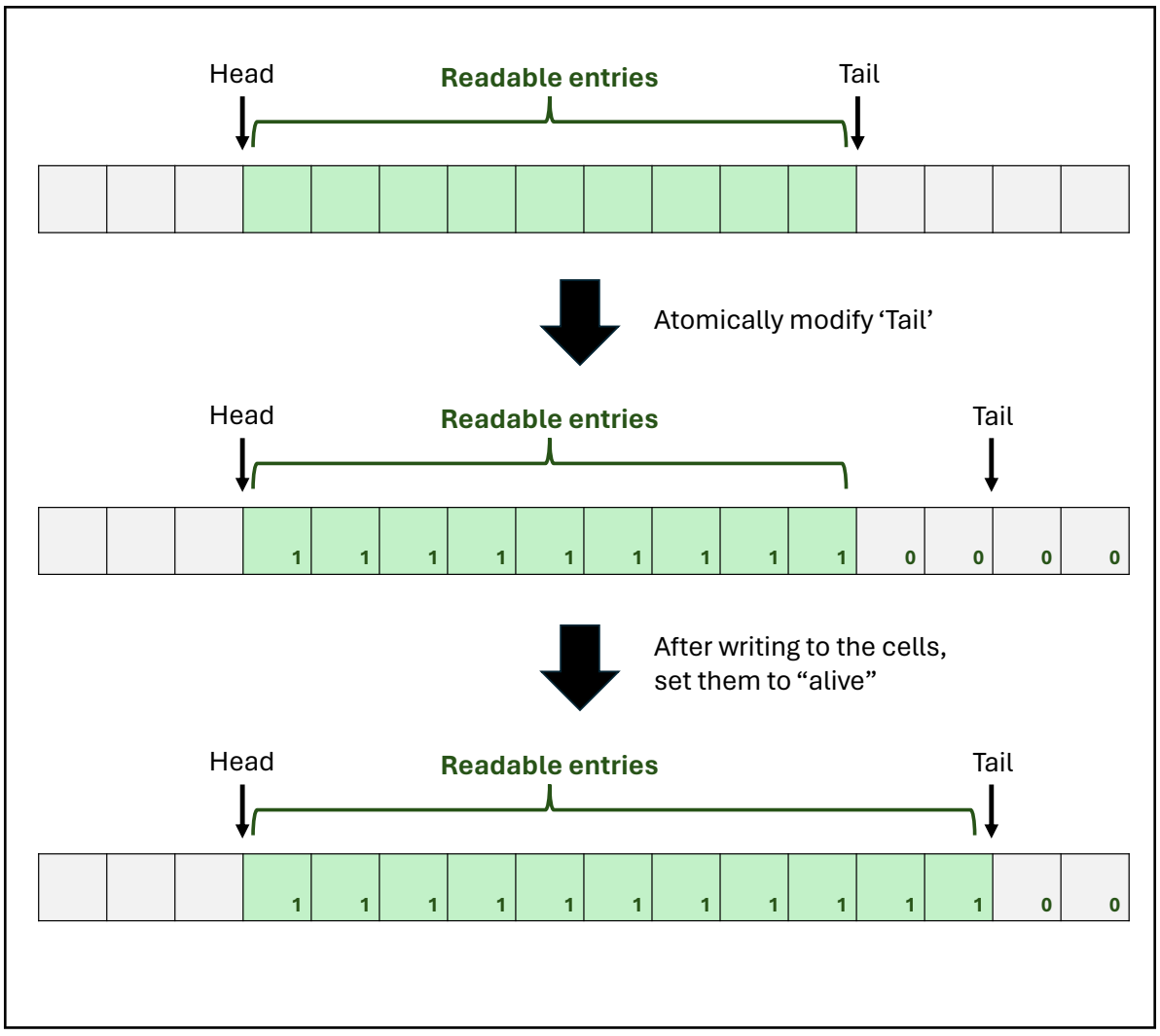


Figure 9.12: **Depiction of inserting messages into the buffer.** ***1.*** *Atomically update the* tail *pointer.* ***2.*** *Mark the entries as alive by setting the "alive" bits. (In general, the meaning of these bits is contextual; on one cycle through the buffer, we set them from 0 to 1 to indicate the entry is alive; on the next cycle, we set them back from 1 to 0 to indicate the entry is alive. In this particular example, we go from 0 to 1, and thus we use 1 to mean "alive.")*

for threads to read the live entries via a guard relationship, `(&combiner) -> &cell::PointsTo`. The `combiner` is a per-node ghost object that tracks the interactions of that node's combiner thread with the message buffer.

Replicas, operations, and the update log

To abstract the global operation of the system, we use a GSM called UnboundedLog(**X**). This abstraction is named as such because it tracks the complete history of all update operations (a contrast to the physical operation of NR, where history is thrown away when the cyclic buffer wraps around). UnboundedLog(**X**) has the following state:

- A log of `UpdateOperation` values, numbered starting at 0.
- A tail, indicating the length of the log. The ghost token for the tail is associated with the atomic containing message buffer's tail.
- Per-node replica states.
- Per-node versions. The ghost tokens for these values are associated with the atomics containing the message buffer's version pointers.
- The `version_upper_bound` field, an upper bound on all node versions. This is used to synchronize update operations with read operations, to make sure that read operations are performed on sufficiently up-to-replicas.
- States for in-progress queries and updates (indexed by `RequestId`s).
- States for the per-node combiner threads.

It is worth noting that the state here "overlaps" some of the state used in CyclicBuffer, but this is not a problem; we can always just associate multiple ghost tokens with the same physical atomic location.

Now, how does the implementation manage all these ghost tokens?

- Ghost tokens `tail`, `local_versions`, and `version_upper_bound`, are all associated with various atomics.
- The `replica` ghost tokens are protected by the same lock that protects the replica.
- The `log` ghost tokens are stored in the CyclicBuffer system; just like the cell memory, we can guard, deposit, and withdraw the log tokens. (The ghost tokens for old log entries are eventually dropped, but they are not *removed* from the UnboundedLog(**X**) system.)
- The ghost tokens for updates and queries are held onto by the threads performing the relevant processing. Each `query` or `update` token starts in the `QueryState::Init` or `UpdateState::Init` state with an operation to be performed; each one progresses through multiple steps to reach the `Done` state, which contains the response.
- The `combiner` ghost token is protected by the same lock that protects the replica. Whenever the lock is not held, it is kept in the `Idle` state, indicating that the combiner is inactive. This is important as we discuss later in §9.5.2. A thread can become the combiner thread by taking this lock and taking ownership of the combiner token. There, they proceed through a series of states and and ultimately return to the `Idle` state when combining is done.

```
type LogIdx = nat;

tracked struct StoredType<DT: Dispatch> {
    pub cell_perms: cell::PointsTo<Option<ConcreteLogEntry<DT>>>,
    pub log_entry: Option<UnboundedLog::log<DT>>, // Ghost token from UnboundedLog
}

ghost enum ReaderState<DT: Dispatch> {
    Starting { start: LogIdx }
    Range { start: LogIdx, end: LogIdx, cur: LogIdx },
    Guard { start: LogIdx, end: LogIdx, cur: LogIdx, val: StoredType<DT> },
}

ghost enum CombinerState<DT: Dispatch> {
    Idle,
    Reading(ReaderState<DT>),
    AdvancingHead { /* ... */ },
    AdvancingTail { /* ... */ },
    Appending { /* ... */ },
}

CyclicBuffer<DT: Dispatch> {
    fields {
        #[sharding(constant)]    pub unbounded_log_instance:
                                     UnboundedLog::Instance::<DT>,
        #[sharding(constant)]    pub cell_ids: Seq<CellId>,
        #[sharding(constant)]    pub buffer_size: LogIdx,
        #[sharding(constant)]    pub num_replicas: nat,
        #[sharding(variable)]    pub head: LogIdx,
        #[sharding(variable)]    pub tail: LogIdx,
        #[sharding(map)]         pub local_versions: Map<NodeId, LogIdx>,
        #[sharding(storage_map)] pub contents: Map<LogicalLogIdx, StoredType<DT>>,
        #[sharding(map)]         pub alive_bits: Map<LogIdx, bool>,
        #[sharding(map)]         pub combiner: Map<NodeId, CombinerState<DT>>
    }

    property!{
        combiner_state_reading_guards(node_id: NodeId) {
            have combiner >= [
                node_id => let CombinerState::Reading(
                    ReaderState::Guard{ start, end, cur, val })
            ];
            guard contents >= [ cur as int => val ];
        }
    }

    // + 15 transitions
}
```

Figure 9.13: **VerusSync fields for NR's CyclicBuffer.**

Performing an update

All updates are performed by the combiner thread, which means that update operations and associated ghost tokens need to be communicated between the requesting thread and the combiner thread. Operations are communicated through some atomic-protected cells; this is proved correct via a VerusSync system FlatCombine, which is also responsible for ferrying the update operation ghost tokens to and fro. FlatCombine is relatively simple compared to the other systems in NR, so I will not say much more about it here.

The combiner takes a write-lock on the replica. This allows it to access both the replica state and the `combiner` ghost token. Using the `combiner` token, the combiner thread can initiate the combiner phase, where it performs **execute**:

- The combiner increments the `tail` index. Recall from earlier that this is an important step in operating the message buffer. It is *also* an important step in the UnboundedLog, where we update the `tail` ghost token and produces new `log` tokens.
- The combiner writes messages into the buffer as described earlier. In doing so, the combiner deposits the `log` ghost tokens.
- The combiner processes all new messages in the queue. To do so, it can read all the relevant cells using `&cell::PointsTo` permissions that are guarded by the CyclicBuffer system. In the same way, it can access the guarded `&log` tokens, allowing it to perform relevant steps in the UnboundedLog—namely, updating its node's replica state. It also updates the `update` tokens as it encounters the corresponding updates.
- It updates the node's `version` as well as the global `version_upper_bound`.
- With the `combiner` token returned to its `Idle` state, the combiner releases the replica's lock.

Performing a query

Performing a query is relatively simple in comparison.

- The thread checks `version_upper_bound` to learn the minimal version their replica needs to be at to perform the query. The thread waits for the replica to reach that version (or performs the updates itself).
- The thread takes a read-lock on the replica. In doing so it gets access to the (read-only) replica state, and it gets the `&combiner` token in the `Idle` state. With this, it can perform the relevant query against the replica and update the `query` token with the result.

The Replica Lock

NR has its own implementation of a reader-writer lock, whose main optimization is that it has multiple read-counters spread across different cache lines. In fact, this was an optimization used by SplinterCache as well, meaning NR's replica lock is strictly less complicated than the lock we already discussed in the previous section. NR uses a VerusSync system NRRwLock, but there is not much more to say about it.

```
type NodeId = nat;

ghost struct LogEntry<DT: Dispatch> {
    pub op: DT::WriteOperation,
    pub node_id: NodeId,
}

ghost enum QueryState<DT: Dispatch> {
    Init { op: DT::ReadOperation },
    VersionUpperBound { /* ... */ },
    ReadyToRead { /* ... */ },
    Done { ret: DT::Response, /* ... */ },
}

ghost enum UpdateState<DT: Dispatch> {
    Init { op: DT::WriteOperation },
    Placed { /* ... */ },
    Applied { /* ... */ },
    Done { ret: DT::Response, /* ... */ },
}

ghost enum CombinerState {
    Idle,
    Placed { /* ... */ },
    LoadedLocalVersion { /* ... */ },
    Loop { /* ... */ },
    UpdatedVersion { /* ... */ },
}

UnboundedLog<DT: Dispatch> {
    fields {
        #[sharding(constant)] pub num_replicas: nat,
        #[sharding(map)]      pub log: Map<LogIdx, LogEntry<DT>>,
        #[sharding(variable)] pub tail: nat,
        #[sharding(map)]      pub replicas: Map<NodeId, DT::View>,
        #[sharding(map)]      pub local_versions: Map<NodeId, LogIdx>,
        #[sharding(variable)] pub version_upper_bound: LogIdx,
        #[sharding(map)]      pub query: Map<ReqId, QueryState<DT>>,
        #[sharding(map)]      pub update: Map<ReqId, UpdateState<DT>>,
        #[sharding(map)]      pub combiner: Map<NodeId, CombinerState>
    }

    // + 13 transitions
    //    3 related to queries
    //    10 related to updates / the combiner
}
```

Figure 9.14: **VerusSync fields for NR's UnboundedLog.**

State machine refinement

Recall our goal is to show linearizability. To do this we need to establish a refinement between UnboundedLog(**X**) and the Linearized(**X**) system described above. Also recall (Challenge NR-3) that NR has future-dependent linearization points. Unfortunately, future-dependence cannot be handled by pure state-based refinements (Definition 1).

To make this more tractable, we divide the process into two steps. First, we create an intermediate abstraction, NRSimple(**X**). We show that UnboundedLog(**X**) refines NRSimple(**X**), then show that NRSimple(**X**) refines Linearized(**X**).

For the first step, that UnboundedLog(**X**) refines NRSimple(**X**), we use a state-based refinement setup. This refinement necessitates some pretty involved invariants, but because it is a state-based refinement, it is still fairly tractable.

For the second step, that NRSimple(**X**) refines Linearized(**X**), we need to handle the future-dependencies, so we must used a trace-based argument rather than a state-based argument. Fortunately, NRSimple(**X**) is, well, simple, which makes it relatively easy to reason about the traces and identify the linearization points.

NRSimple description

NRSimple's state consists of **(i)** all in-flight operations (queries and updates), **(ii)** a log of update operations, and **(iii)** a integer *version* pointing somewhere in the log.

Queries and updates are multi-step processes. The system nondeterministically advances these processes in some arbitrary interleaving. The system also nondeterministically increments the *version*, which is necessary for the operations to complete. The atomic steps of an update are:

- An update is requested with some `WriteOperation`.
- The `WriteOperation` is appended to the log at some point *idx*.
- At some point when *version* $>$ *idx*, the update completes.

The atomic steps of a query are:

- A query is requested with some `ReadOperation`.
- The current *version* is recorded as *lowerBoundVersion*.
- The query completes, returning some value r which is the result of performing the read operation at the data structure state version i (i.e., the version after the applying the first i entries of the log) for some i where *lowerBoundVersion* $\leq$ *idx* $<$ *version*. (That is, *idx* is between the *version* recorded in the previous step and the current value of *version*.)

In practice, the value of i chosen in the previous step has to do with the replica state of the node where the query is performed, but this information is lost at NRSimple's level of abstraction. In NRSimple's model, the query is performed against an arbitrary version which is chosen nondeterministically upon query completion.

NRSimple linearization argument

Again, the linearization argument is not too hard to explain in plain English. In short, linearization points are tied to the *version*. Specifically, the linearization point occurs when *version* increments from *idx* to $idx + 1$, where *idx* is given both for update and query operations as described above.

Inspecting the steps above should reveal that the linearization point has to occur between the start and end of each operation. We can also see *why* the linearization points are future-dependent: For a query operation, the *idx* is not determined until the last step of the query, but the linearization point must necessarily have been *before* this point. (This is not merely an artifact of the "information loss" associated with NRSimple; the same basic point remains in UnboundedLog.)

9.3 Case Study III: Mimalloc

9.3.1 Specification and TCB

For the memory allocator, we'll be using a 'normal' Hoare-style Verus spec as the primary specification, rather than a GSM refinement theorem as in the last two case studies. Why not use a GSM? Well, the plain answer is just that it never came up. Nothing in this project felt it would benefit from a GSM or a tall refinement stack. This is fortunate, since the Challenge Mem-1 challenge, which says that we need to safely maintain a distinction between memory belonging to the allocator and memory given to the client, essentially demands that our main spec be based on passing ghost permissions. This isn't something we could have sensibly done with a GSM spec.

So what *is* our spec, then? The ideal specification of a memory allocator is in fact very simple: `malloc` returns a pointer and some memory permission for the appropriate range of memory (starting at the pointer and extending for the given number of bytes). Meanwhile, `free` requires the caller to return the memory permissions.

In fact, Verus's pointer primitives already have a specification with exactly this flavor. Recall the `PPtr` interface:

```
pub fn alloc(v: V) -> (ptr: PPtr<V>,
    Tracked(perm): Tracked<PointsTo<V>>,
    Tracked(dealloc_perm): Tracked<Dealloc<V>>
)
    ensures
        perm.value() == v,
        perm.pptr() == ptr.id(),
        dealloc_perm.pptr() == ptr.id(),

pub fn free(self,
    Tracked(perm): Tracked<PointsTo<V>>,
    Tracked(dealloc_perm): Tracked<Dealloc<V>>
) -> (out: V)
    requires
        self.id() == perm.pptr(),
        self.id() == dealloc_perm.pptr(),
        perm.value().is_some(),
    ensures
        out == perm.value.unwrap();
```

The interface to our allocator is conceptually very similar, though there are a few differences.

For one thing, `malloc` and `free` operate in bytes, so the input is just an integer, the desired size of the allocation. A larger source of complication is that we need to account for an object to serve as a handle for the allocator itself, since Verus does not have support for globals. In fact, we actually have to deal with *per-thread* handles.

Nonetheless, the specification is small enough that we can present it in its entirety (Figure 9.16). The complicated aspect here is that there is a system for initializing threads. After performing "global initialization" we obtain a bunch of tokens that give us the "right" to instantiate heaps for various thread IDs. The specs for `heap_malloc` and `free` are more straightforward, there are just a bunch of well-formedness connections and conditions making sure everything is correctly associated to the right allocator.

```
ghost type MimInst;
tracked type Global;
tracked type RightToUseThread;
tracked type Local;
type HeapPtr;
tracked type Dealloc;

impl Global {
    spec fn wf(self) -> bool;
    spec fn inst(self) -> MimInst;
    spec fn wf_right_to_use_thread(self, right: RightToUseThread, tid: ThreadId)
        -> bool;
}
impl Local {
    spec fn wf(self) -> bool;
    spec fn inst(self) -> MimInst;
}
impl HeapPtr {
    spec fn is_in(self, local: Local) -> bool;
}
import Dealloc {
    spec fn wf(self) -> bool;
    spec fn inst(self) -> MimInst;
    spec fn ptr(self) -> usize;
    spec fn size(self) -> int;
}

// (Ghost operation)
// Initialize the global identifier that ties all handles together
pub proof fn global_init()
    -> (tracked global: Global,
        tracked rights: Map<ThreadId, Mim::right_to_use_thread>))
    ensures
        global.wf(),
        forall |tid: ThreadId| thread_rights.dom().contains(tid)
          && global.wf_right_to_use_thread(thread_rights[tid], tid)

// Initialize a handle for a single thread
// The `IsThread` token is from the trusted thread library and tells us what
// thread we are on.
pub fn heap_init(Tracked(global): Tracked<Global>,
      Tracked(thread_right): Tracked<RightToUseThread>,
      Tracked(cur_thread): Tracked<IsThread>
) -> (heap: HeapPtr, Tracked(local_opt): Tracked<Option<Local>>)
    requires
        global.wf_right_to_use_thread(thread_right, cur_thread@),
        global.wf(),
    ensures
        heap.heap_ptr.id() != 0 ==>
            local_opt.is_some()
            && local_opt.unwrap().wf()
            && local_opt.unwrap().inst() == global.inst()
            && heap.wf()
            && heap.is_in(local_opt.unwrap())
```

Figure 9.15: **Top-level specification for the memory allocator thread initialization.**

```
// Perform an allocation
pub fn heap_malloc(heap: HeapPtr, size: usize,
    Tracked(local): Tracked<&mut Local>
)
    -> (ptr: PPtr<u8>,
        Tracked(points_to_raw): Tracked<PointsToRaw>,
        Tracked(dealloc): Tracked<Dealloc>))
    requires
        old(local).wf(),
        heap.wf(),
        heap.is_in(*old(local)),
    ensures
        local.wf(),
        local.inst() == old(local).inst(),
        forall |heap: HeapPtr| heap.is_in(*old(local)) ==> heap.is_in(*local),
        dealloc.wf()
        points_to_raw.is_range(ptr.id(), size as int)
        ptr.id() == dealloc.ptr()
        dealloc.inst() == local.inst()
        dealloc.size() == size

// Free an allocation
pub fn free(
    ptr: PPtr<u8>,
    Tracked(user_perm): Tracked<ptr::PointsToRaw>,
    Tracked(user_dealloc): Tracked<Option<Dealloc>>,
    Tracked(local): Tracked<&mut Local>
)
    // According to the Linux man pages, `ptr` is allowed to be NULL,
    // in which case no operation is performed.
    // Therefore, our precondition is conditional on the ptr being non-null
    requires
        old(local).wf(),
        ptr.id() != 0 ==> (
            user_dealloc.is_some()
              && user_dealloc.unwrap().wf()
              && user_perm.is_range(ptr.id(), user_dealloc.unwrap().size())
              && ptr.id() == user_dealloc.unwrap().ptr()
              && old(local).inst() == user_dealloc.unwrap().inst()
        )
    ensures
        local.wf(),
        local.inst() == old(local).inst(),
        forall |heap: HeapPtr| heap.is_in(*old(local)) ==> heap.is_in(*local),
```

Figure 9.16: **Top-level specification for the memory allocator, free and malloc.**

OS memory interface

At a high level, the role of a memory allocator is to bridge the gap between the OS memory interface, which provides coarse-grained allocations at page boundaries, and the user interface of `malloc` and `free`. The OS interface we consider is the Linux `mmap` syscall.

It is relatively straightforward to provide a trusted specification of `mmap` in terms of ghost memory permissions. One slight complication is that we need to account for multiple possible memory states. Mimalloc often reserves memory without marking it writable, which means we need to maintain the right to mark it writable without actually getting points-to permissions for it. To handle this use-case, we add an extra ghost object that tracks the permissions associated with a range of virtual memory.

Thread IDs

The implementation needs to be able to get unique thread IDs, which is more nontrivial than it sounds. Verus, by default, *does* support some basic threading utilities, including a function to get a thread-unique ID by calling the Rust standard library's `std::thread::current().id()`. However, this function actually calls into the memory alloctor internally, so we cannot use it here.

Implementing unique thread IDs both efficiently and in a platform-independent way is somewhat challenging. The easiest way to do it is to use the address of a variable in Thread Local Storage, though this only works on some platforms (as on some platforms, the implementation of TLS itself relies on an allocator). Thus, we implement a trusted `thread_id()` function that does exactly this, albeit with support restricted to Linux.

This aspect does raise some questions about how one might verify cross-platform code with such complex OS interactions in a principled manner, but this is far out of scope here.

9.3.2 Proof overview

Memory permissions and the organization of a segment

Broadly speaking, the way a memory allocator works is as follows:

- The allocator acquires some memory from the OS in a platform-dependent way. On Linux, for example, it would use the `mmap` system call to reserve memory.
- Some of the memory is reserved for allocator-internal metadata, while some of the memory is allocated out to the client.

More specifically, in mimalloc, most memory is organized in entities called *segments*. The start of the segment is reserved for *segment header*, while the remainder of the segment is divided into *pages*. Each page is divided into blocks, which can be allocated to the client. The unallocated blocks are organized into a linked lists called *free lists*, so allocating works by popping a block off of a free list, whereas freeing works by pushing a block onto a free list.

Furthermore, the division of a page into segments is somewhat complex: the way this works is that the segment is divided into *slices*, and a page is allocated out of a contiguous range of slices. See Figure 9.17 for a detailed picture.

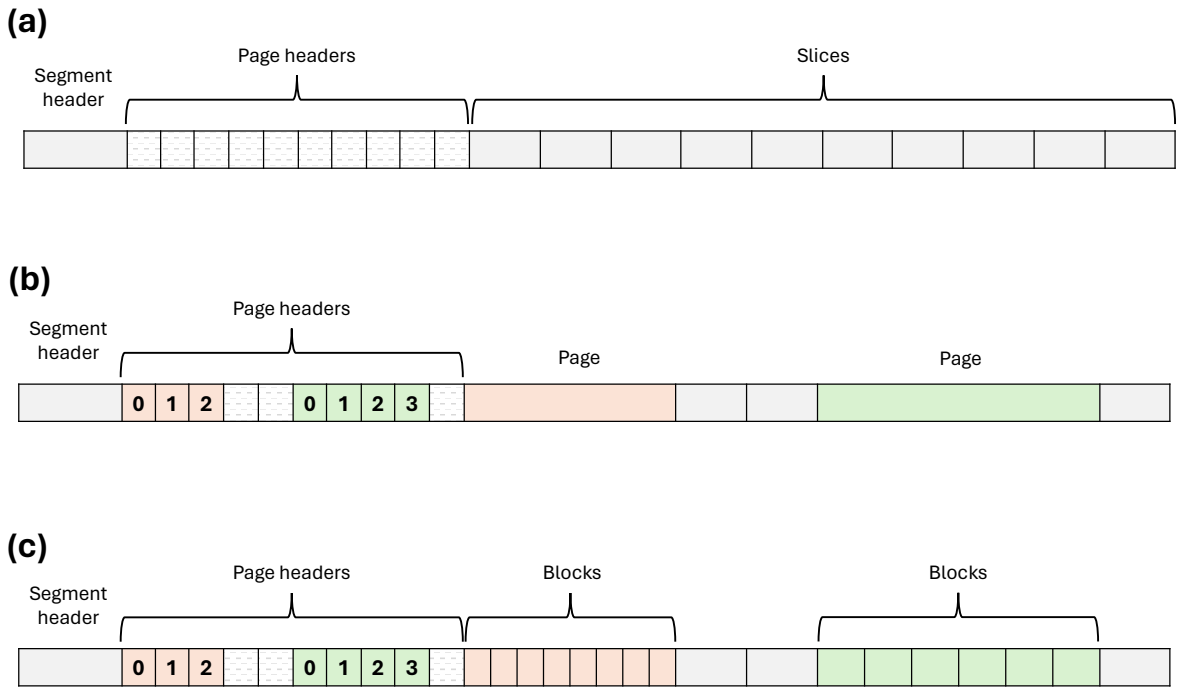


Figure 9.17: **The layout of a segment in mimalloc and in Verus-mimalloc.** ***(a)*** *The start of a segment contains the "segment metadata": a segment header and a number of page headers. The rest of the segment is divided into slices; there is one page header per slice.* ***(b)*** *Pages are formed by coalescing slices; in the example, we have one page made of 3 slices and one page made of 4 slices. Each page corresponds to a number of page headers; the first of these page headers is the "primary" page header for the page. Each page header has an "offset" so from any slice you can always find the primary page header.* ***(c)*** *Each page is divided into blocks which may be allocated to the user. The block size varies between pages, but it is fixed within a given page.*

9.3.3 Thread local data structures and concurrency

The operation of the memory is allocator is mostly thread-local, though the word "mostly" pulls a lot of weight here. More specifically, each thread has its own "heap" which comprises a bunch of segments. Whenever `malloc` is called, the allocator pops a block from one of the segments belonging to the current thread. Hence, it primarily accesses data that is specific to that one thread. However, it is possible for a `free` operation to be performed from a thread different from which the original allocation was performed.

Specifically, `free(p)` works as follows:

- Compute the start of the segment containing `p`. (Segments are always aligned, so we just round down to the segment size.)
- Find the *page* that contains this allocation. To do this:
  - Calculate the *slice* containing `p` (again easy to do by rounding down).
  - Look up the `offset` for the page header corresponding to this slice.
  - Subtract the offset to get the primary page header for the page corresponding to this slice.
- Look up the thread ID from the segment header to determine if the allocation is thread-local or not.
  - If thread-local:
    - Push the block onto the page's thread-local free list.
  - If not thread-local:
    - Push the block onto the page's *atomic* free list, a special free list which is only accessed via atomic operations (e.g., compare-and-swap).

We can see this introduces quite a bit of complexity. First of all, certain fields need to be accessible to the other threads. Secondly, any proof of the `free(p)` operation needs to make a number of deductions:

- First, we need to use the fact that an allocation exists at `p` to deduce the existence of the segment and that it is safe to access the thread ID field from the segment header.
- Likewise, we need to deduce that it is safe to access the two page headers—both the one for the pointer's slice and the one obtained by subtracting the `offset`.
- Finally, if the thread ID matches our own, then we need to deduce that it is safe to access the thread-local parts of the segment (e.g., the thread-local free list).

Actually, there is yet another complication: while performing a non-local free, it is sometimes necessary (to avoid leaks, though not for safety) to insert the block in a "centralized" atomic free list which is located on the heap header rather than inserting it into the usual page header. This is called a *delayed free*. The mimalloc technical report [47] details the rationale for delayed frees—it has to do with something called "the full list" which I am mostly eliding from the discussion here. At any rate, when this scenario occurs, the thread proceeds by looking up the pointer to the heap header from the page header; it then accesses the atomic free list on the heap header. While doing so, it uses a locking mechanism to ensure that the heap remains valid

```rust
#[repr(C)]
pub struct Page {
    pub count: PCell<u32>,   // number of slices in this page
    pub offset: u32,         // offset of this slice compared to the "main" page header
    pub inner: PCell<PageInner>,  // free lists and other thread-local fields
    pub xthread_free: ThreadLLWithDelayBits,  // atomic free list
    pub xheap: AtomicHeapPtr,     // pointer to the heap containing this page
    pub prev: PCell<PPtr<Page>>,  // previous page in the page queue
    pub next: PCell<PPtr<Page>>,  // next page in the page queue
    pub _padding: usize,
}

pub tracked struct PageSharedAccess {
    pub tracked points_to: ptr::PointsTo<Page>,
}

pub tracked struct PageLocalAccess {
    pub tracked count: cell::PointsTo<u32>,
    pub tracked inner: cell::PointsTo<PageInner>,
    pub tracked prev: cell::PointsTo<PPtr<Page>>,
    pub tracked next: cell::PointsTo<PPtr<Page>>,
}
```

Figure 9.18: **Example of the "local/shared" split for a page header.** *When a* `PPtr<Page>` *is combined with the ghost reference* `&PageSharedAccess`*, the code can obtain a reference* `&Page`*. This gives them, for example: read-only access to the* `offset` *field, or concurrent access to the* `xthread_free` *field, which contains the atomic free list. However, there is no way to access the interiors of the* `PCell` *fields. On the other hand, if we further have ownership of* `PageLocalAccess`*, then we* do *have permission to access the interiors, thus allowing us to access and modify the* `count`*,* `inner`*,* `prev`*, and* `next` *data, that is, all the "thread local" state.*

for this operation (to avoid races when the heap is destroyed during thread cleanup[2]).

In the end, all this comes down to the need to reason about both thread-local access and shared access to these data structures. Therefore, our first step is to create a split for these concepts: We create *two* ghost objects that represent access to the page, `PageLocalAccess` and `PageSharedAccess`. The shared reference `&PageSharedAccess` gives all threads the concurrent access they need, while further ownership of `PageLocalAccess` gives full access to all the threads that are meant to be accessed thread-locally. Figure 9.18 shows how these are constructed. We do likewise for segment headers and the heap header.

Now, each thread can hold on to its own relevant `PageLocalAccess` ghost objects. The challenge is finding the right way to share the `&PageSharedAccess` ghost objects so all the threads can access them. Coordinating access to the shared reference of a ghost state is exactly what a storage protocol is useful for.

Our VerusSync storage protocol, here called Mim, models the relationship between threads, heaps, segments, pages, and blocks. Mim has multiple storage fields allowing the user to store

[2]Actually, thread clean-up is currently unimplemented in Verus-mimalloc, but we still account for its possibility in our proof.

`PageSharedAccess`, `SegmentSharedAccess`, and `HeapSharedAccess` objects in Mim, and there are a number of guarding relationships that allow shared access to these ghost objects.

Here are some of the highlights of Mim:

- When a thread initializes its heap, it deposits the `HeapSharedAccess` into Mim, and when it destroys a heap, it withdraws it.
- When a thread initializes a new segment, it deposits the `SegmentSharedAccess` into Mim, and when it destroys a segment, it withdraws it.
- When a thread activates a page, i.e., when it fixes the boundaries of the page, fixes the block size and initializes the free lists, it deposits the `PageSharedAccess` into Mim. When it deactivates the page, it withdraws it.
- Mim has a token type called `ThreadLocal`, of which there is one per thread. This token gives each thread the authority to modify its segments, pages, etc.
- For each activated page, Mim creates `Block` tokens that represent possible block allocations.
- We have guards so that a thread will always have shared access to its own pages, segments, and heap:
  - `(&ThreadLocal) -> &PageSharedAccess`
  - `(&ThreadLocal) -> &SegmentSharedAccess`
  - `(&ThreadLocal) -> &HeapSharedAccess`
- We have guards so that any thread performing a `free` will be able to access the relevant headers:
  - `(&Block) -> &SegmentSharedAccess`
  - `(&Block) -> &PageSharedAccess` (for the pointer's slice)
  - `(&Block) -> &PageSharedAccess` (for the slice after subtracting the offset)
- There is a token called `DelayActor` that is used during a delayed free. Owning this token corresponds to the "lock" used to make sure that the heap and page remain valid for the duration of the operation. We need the guards:
  - `(&DelayActor) -> &HeapSharedAccess`
  - `(&DelayActor) -> &PageSharedAccess`

With all this established, we can now see how `free` falls into place. Let's walk through the implementation of `free` again, this time describing the ghost steps:

- Compute the start of the segment containing `p`. Use `&Block` to guard `&SegmentSharedAccess`.
- Find the *page* that contains this allocation. To do this:
  - Calculate the *slice* containing `p`. Use `&Block` to guard `&PageSharedAccess`.
  - Using the `&PageSharedAccess`, read the `offset` for the page header corresponding to this slice.
  - Subtract the offset to get the primary page header for the page corresponding to this slice. Use `&Block` to guard `&PageSharedAccess` for this slice as well.
- Look up the thread ID from the segment header to determine if the allocation is thread-local

or not.

- If thread-local:
  - Get all the relevant permissions from the thread-local ghost state, and use `&ThreadLocal` to guard `&PageSharedAccess` for the relevant page. Since we are on the right thread, we have ownership of `PageLocalAccess` as well.
  - Push the block onto the page's thread-local free list. This relinquishes the `Block` token.
- If not thread-local (and not using delayed-free):
  - Using the `&PageSharedAccess`, concurrently access the atomic free list.
- If not thread-local, but using delayed-free:
  - Using the `&PageSharedAccess`, read the page's pointer to the Heap.
  - Take the lock and obtain the `&DelayActorToken`.
  - Use `&DelayActorToken` to guard `&HeapSharedAccess`
  - Concurrently access the Heap's atomic free list and insert the block. This relinquishes the `Block` token.
  - Use `&DelayActorToken` to guard `&PageSharedAccess`. (We need to do this since we lost the `Block` token.)
  - Release the lock. This relinquishes the `Block` token.

Relinquishing ghost state and atomic operations It is interesting to observe that many of these actions perform atomic operations on memory that simultaneously give up the ability to access that memory. Initially, it might not be obvious if this actually works out, but it does. We end up with something like this:

```
1  struct AtomicU64<G> {
2      pub patomic: PAtomicU64,    // Verus atomic primitive
3      pub atomic_inv: Tracked<AtomicInvariant<_, (PermissionU64, G), _>,
4  }
5
6  pub example() {
7      // We have some token G (e.g., like Block or DelayActorToken) ...
8      let tracked g: G = /* ... */;
9
10     // ... Which guards a reference to some atomic.          /* lifetime of &g */
11     let atomic: &AtomicU64<G> = guard(Tracked(&g));                              -+
12                                                                                  |
13     open_atomic_invariant!(&atomic.atomic_inv => pair_permission_g => {          |
14         atomic.patomic.store(some_value, Tracked(&mut pair_permission_g.0));  -+
15         proof {
16             pair_permission_g.1 = g;                          /* g is moved */
17         }
18     });
19 }
```

The borrow of `&g` on line 11 extends through line 14, where the variable `atomic` is last used. This means the borrow expires before we try to move `g` on line 16. Thus everything works out.

This pattern does crucially rely on non-lexical lifetimes, though, and it forces us to inline certain functions that I would prefer to abstract out.

Furthermore, there was actually a problem with this in an earlier version of Verus. Specifically, the earlier version had an additional restriction, and as a result, the use of `&atomic.atomic_inv` on line 13 caused the lifetime of the `&g` borrow to be extended all the way to the block close, on line 18. This *was* problematic, since then we could not move `g` on line 16. This was possible to work around, though in the end, we ended up removing this restriction entirely (as discussed in §6.3.3).

9.4 Case Study IV: Reference-Counted Smart Pointers

9.4.1 Specification and TCB

Recall that one of our goals (Challenge RC-2) is to handle the distinction between thread-safe reference counts (such as Rust's `Arc`) and single-threaded reference counts (`Rc`). In this section, we consider simplified implementations of both `Arc` and `Rc`. Specifically, we focus on four essential operations:

- `new`, which creates a new allocation with a single handle to it.
- `clone`, which creates a new handle to an existing handle.
- `drop`, which destroys a handle (and frees its allocation if there are no other handles).[3]
- `borrow`, used to access the object stored in an allocation. Since access to the underlying object may be shared, this naturally must be done via a shared reference, `&T`.

Figure 9.19 shows the type signatures and desired specifications for these functions. Observe that these specifications effectively allow the user to reason about the `Rc<T>`/`Arc<T>` object "as if" it were simply a `T` object. To do so they merely need to reason about the specification value, `self.view()`.

An astute reader may notice that nothing in our specification actually requires us to free the memory when the reference count hits zero. Nonetheless, the implementation does in fact do that, and this fact naturally complicates the proof of safety.

9.4.2 The implementations

Unverified implementations (i.e., implementations in "normal Rust") are shown for `Rc` in Figure 9.20 and for `Arc` in Figure 9.21. These are not the "official" implementations from the standard library (recall that we're only using `SeqCst` memory ordering, for example), though they are pretty close.

Let us look at `Rc` first, which is the simpler one. Really there isn't much to it. The counter is stored in an `UnsafeCell`, an interior mutability type, so it can be accessed by multiple handles. When `clone` is called, we increment the counter. When `drop` is called, we decrement the counter. If the counter hits zero, we call a deallocation routine on the global allocator.

Interestingly, the overflow-check in `clone` *is* a part of the official implementation, not just a shortcut we're taking. At first, it seems like one could make an argument like: "This integer can never overflow because the total number of pointer handles in existence can never exceed the size of the virtual address space," and that as a result, the overflow check can be omitted. However, it's actually possible to "lose" handles without destroying them (e.g., through `std::mem::forget`) so the overflow-check really is required for memory safety.

Inspection shows that the implementation for `Arc` is broadly similar, with the main difference being the way the reference count field is accessed: here, it is done with atomic instructions.

[3] Ordinarly, `drop` is implemented via the `Drop` trait and is called automatically as a convenience. In Rust, the trait function `drop` has signature `drop(&mut self)`, but it is a bit simpler to work with `drop(self)`, which consumes its argument, as we do in this section.

```
type Rc<T>

impl<T> !Sync for Rc<T> {}
impl<T> !Send for Rc<T> {}

impl<T> Rc<T> {
    pub spec fn view(&self) -> T;

    pub fn new(t: T) -> (rc: Self)
        ensures rc.view() == t

    pub fn clone(&self) -> (rc: Self)
        ensures rc.view() == self.view()

    pub fn drop(self)

    pub fn borrow(&self) -> (t: &T)
        ensures *t == self.view()
}
```

```
type Arc<T>

impl<T: Send + Sync> Sync for Arc<T> {}
impl<T: Send + Sync> Send for Arc<T> {}

impl<T> Arc<T> {
    // ... similar to Rc<T>
}
```

Figure 9.19: **Verus specifications for `Rc` and `Arc`.**

Because the compare-exchange function can fail, it has to be wrapped in a loop. We cannot use `fetch_add` because we need to check for overflow.[4]

The differences in the implementation mean that `Arc` is thread-safe while `Rc` is not. These are reflected in their type signatures (Figure 9.19), and in particular, in the way they do or not implement the marker traits `Send` and `Sync` (§3.4.6):

- `Rc<T>` implements neither `Send` nor `Sync` since it is not thread-safe. Since it implements neither marker trait, it cannot be used across threads.
- `Arc<T>` implements both `Send` and `Sync` (provided that `T` does). This allows it to be used across threads.

9.4.3 Verified implementations

The main effort is to put together the right types; then all else will follow. Naturally, we will use a `PCell` for the counter, replacing the `UnsafeCell`. To make sure any handle will be able to access it, we put the `cell::PointsTo` in a `LocalInvariant` which is shared by all the handles. We can also tie the value to some `counter` ghost token, as in counting permissions. And of course, we'll have a `reader` guard token.

```
// Struct that the pointer points to.
struct InnerRc<S> {
    // Counter
    pub rc_cell: PCell<u64>,
    // Underlying use object
    pub s: S,
}

// All the memory permissions to access and deallocate \code{InnerRc}.
type MemPerms<S> = (ptr::PointsTo<InnerRc<S>>, ptr::Dealloc<InnerRc<S>>);

// Ghost tokens that go in the invariant.
tracked struct GhostStuff<S> {
    pub tracked rc_perm: cell::PointsTo<u64>,
    pub tracked rc_token: RefCounterTokens::counter<MemPerms<S>>,
}

struct VerusRc<S> {
    // The actual physical pointer
    pub ptr: PPtr<InnerRc<S>>,

    // Instance for the VerusSync system
    pub inst: Tracked<RefCounterTokens::Instance<MemPerms<S>>>,
    // Token that gives us access to &InnerRc<S> and &S
    pub reader: Tracked<RefCounterTokens::reader<MemPerms<S>>>,
    // Invariant giving us access to the counter.
    pub inv: Tracked<Duplicable<LocalInvariant<CellId, GhostStuff<S>, Pred>>>,
}
```

[4]The standard library implementation uses `fetch_add` anyway, and a comment in the source code argues that an "exceedingly unlikely" situation would have to occur in order for it to be incorrect. I will not attempt to evaluate this claim; I will only observe that we have no formal means of making such an argument. Thus, our implementation uses a loop.

```rust
// Implementation of Rc

struct InnerRc<T> {
    rc_cell: std::cell::UnsafeCell<u64>,
    t: T,
}

struct Rc<T> {
    ptr: *mut InnerRc<T>,
}

impl<T> Rc<T> {
    fn new(t: T) -> Self {
        // Allocate a new InnerRc object, initialize the counter to 1,
        // and return a pointer to it.
        let rc_cell = std::cell::UnsafeCell::new(1);
        let inner_rc = InnerRc { rc_cell, t };
        let ptr = Box::leak(Box::new(inner_rc));
        Rc { ptr }
    }

    fn clone(&self) -> Self {
        unsafe {
            // Increment the counter.
            // If incrementing the counter would lead to overflow, then abort.
            let inner_rc = &*self.ptr;
            let count = *inner_rc.rc_cell.get();
            if count == 0xffffffffffffffff {
                std::process::abort();
            }
            *inner_rc.rc_cell.get() = count + 1;
        }

        // Return a new Rc object with the same pointer.
        Rc { ptr: self.ptr }
    }

    fn drop(self) {
        unsafe {
            // Decrement the counter.
            let inner_rc = &*self.ptr;
            let count = *inner_rc.rc_cell.get() - 1;
            *inner_rc.rc_cell.get() = count;

            // If the counter hits 0, drop the `T` and deallocate the memory.
            if count == 0 {
                std::ptr::drop_in_place(&mut (*self.ptr).t);
                std::alloc::dealloc(self.ptr as *mut u8,
                    std::alloc::Layout::for_value(&*self.ptr));
            }
        }
    }

    fn borrow(&self) -> &T {
        unsafe {
            &(*self.ptr).t
        }
    }
}
```

Figure 9.20: **Unverified implementation of `Rc`.**

```rust
// Implementation of Arc

struct InnerArc<T> {
    arc_cell: std::sync::atomic::AtomicU64,
    t: T,
}

struct Arc<T> {
    ptr: *mut InnerArc<T>,
}

impl<T> Arc<T> {
    fn new(t: T) -> Self {
        // Allocate a new InnerArc object, initialize the counter to 1,
        // and return a pointer to it.
        let arc_cell = std::sync::atomic::AtomicU64::new(1);
        let inner_arc = InnerArc { arc_cell, t };
        let ptr = Box::leak(Box::new(inner_arc));
        Arc { ptr }
    }

    fn clone(&self) -> Self {
        unsafe {
            // Increment the counter.
            // If incrementing the counter would lead to overflow, then abort.
            let inner_arc = &*self.ptr;
            loop {
                let count = inner_arc.arc_cell.load(Ordering::SeqCst);
                if count == 0xffffffffffffffff {
                    std::process::abort();
                }
                let res = inner_arc.arc_cell.compare_exchange_weak(
                    count, count + 1, Ordering::SeqCst, Ordering::SeqCst);
                if res.is_ok() {
                    break;
                }
            }
        }

        // Return a new Arc object with the same pointer.
        Arc { ptr: self.ptr }
    }

    fn drop(self) {
        unsafe {
            // Decrement the counter.
            let inner_arc = &*self.ptr;
            let count = inner_arc.arc_cell.fetch_sub(1, Ordering::SeqCst);

            // If the counter hits 0, drop the `T` and deallocate the memory.
            // (`count` currently stores the value from before the decrement)
            if count == 1 {
                std::ptr::drop_in_place(&mut (*self.ptr).t);
                std::alloc::dealloc(self.ptr as *mut u8,
                    std::alloc::Layout::for_value(&*self.ptr));
            }
        }
    }

    fn borrow(&self) -> &T {
        unsafe {
            &(*self.ptr).t
        }
    }
}
```

Figure 9.21: **Unverified implementation of `Arc`.**

The `Duplicable` type is a library helper that lets us share the `LocalInvariant` between the pointer handles. By using `Duplicable`, we give up the ability to call `into_inner` on the invariant, but we don't need that here.

The RefCounterTokens VerusSync system we use is in Figure 9.22. It is very similar to counting permissions, and we use it for both `Rc` and `Arc`.

Now writing the verified implementation is pretty straightforward:

- We can use the `reader` object to access `&InnerRc` whenever we want.
- To access the counter, we just open the invariant. This lets us both manipulate the physical counter and access the ghost `counter` token.
- Now we can destroy the `reader` token or create new ones.
- When destroying the last one, we get access to the `MemPerms` which gives us the ability to release the allocation.

The implementation for `Arc` is basically the same, except that we use `AtomicInvariant` instead of `LocalInvariant`, which of course forces us to use atomic instructions.

Marker traits The `Send` and `Sync` marker traits just work out without us needing to do anything special. `LocalInvariant<...>` is not `Sync`, so `Duplicable<LocalInvariant<...>>` is neither `Send` nor `Sync`. (The library feature `Duplicable` is itself implemented with a storage protocol, so it has the marker trait behavior of storage protocols.) That means `VerusRc` is neither `Send` nor `Sync`.

If you think about, though, it couldn't have been anything different. Our code is verified, and anything else would have been unsound. Ergo, this was the only thing it could have been.

But what about `Arc`? Is it correctly permissive? Indeed, it is. First of all, we know that `AtomicInvariant<...>` is both `Send` and `Sync`. Thus, `Duplicable<AtomicInvariant<...>>` is both `Send` and `Sync`, as expected.

Variance of the type parameter Unfortunately, while the marker traits "just work out," the same can not be said of the type parameter variance. It ought to be sound for the `Rc<T>` and `Arc<T>` to be *covariant* in their type parameter `T`; however, this is not the case with the implementation we have shown here. The problem is that `RefCounter::Instance` and `LocalInvariant` are both non-variant in their type parameters. Resolving this is left for future work.

Recursive types We can, in fact, use our verified types in recursive data structures, like so:

```
enum Sequence<V> {
    Nil,
    Cons(V, VerusRc<Sequence<V>>),
}
fn main() {
    let nil = VerusRc::new(Sequence::Nil);
    let a7 = VerusRc::new(Sequence::Cons(7, nil.clone()));
    let a67 = VerusRc::new(Sequence::Cons(6, a7.clone()));
}
```

Verus accepts this because `S` never appears in a negative position in the definition of `VerusRc<S>`.

```
RefCounterTokens<Perm> {
    fields {
        #[sharding(variable)]       pub counter: nat,
        #[sharding(storage_option)] pub storage: Option<Perm>,
        #[sharding(multiset)]       pub reader: Multiset<Perm>,
    }

    init!{ initialize_empty() {
        init counter = 0;
        init storage = Option::None;
        init reader = Multiset::empty();
    }}

    transition!{ do_deposit(x: Perm) {
        require(pre.counter == 0);
        update counter = 1;
        deposit storage += Some(x);
        add reader += {x};
    }}
    transition!{ do_clone(x: Perm) {
        have reader >= {x};
        add reader += {x};
        update counter = pre.counter + 1;
    }
    transition!{ dec_basic(x: Perm) {
        require(pre.counter >= 2);
        remove reader -= {x};
        update counter = (pre.counter - 1) as nat;
    }}
    transition!{ dec_to_zero(x: Perm) {
        remove reader -= {x};
        require(pre.counter < 2);
        assert(pre.counter == 1);
        update counter = 0;
        withdraw storage -= Some(x);
    }}

    property!{ reader_guard(x: Perm) {
        have reader >= {x};
        guard storage >= Some(x);
    }}

    #[invariant]
    pub fn reader_agrees_storage(&self) -> bool {
        forall t: Perm self.reader.count(t) > 0 ==> self.storage == Option::Some(t)
    }

    #[invariant]
    pub fn counter_agrees_storage(&self) -> bool {
        self.counter == 0 ==> self.storage.is_None()
    }

    #[invariant]
    pub fn counter_agrees_storage_rev(&self) -> bool {
        self.storage.is_None() ==> self.counter == 0
    }

    #[invariant]
    pub fn counter_agrees_reader_count(&self) -> bool {
        self.storage.is_Some() ==>
            self.reader.count(self.storage.get_Some_0()) == self.counter
    }
}
```

Figure 9.22: **VerusSync for `Rc` and `Arc`.**

9.5 Evaluation

Our evaluation focuses on the three major case studies. We evaluate on three axes:

- Was what we did realistic?
- What did we learn and gain by doing it?
- How much effort was it?

9.5.1 Was what we did realistic?

Here, we present benchmarks for our three major case studies to see how "realistic" they are. Recall that each major case study is based on an existing system, which is already assumed to be highly performant. Therefore, the main objective of this section is the comparison of our verified version with the original.

Before diving in, I would like to acknowledge the efforts of everybody involved in the benchmarking process: Alex Conway for SplinterCache, and Reto Achermann, Ryan Stutsman, and Gerd Zellweger for Node Replication. I would also like to acknowledge Reto Achermann for driving the completion of the Verus NR port.

SplinterCache

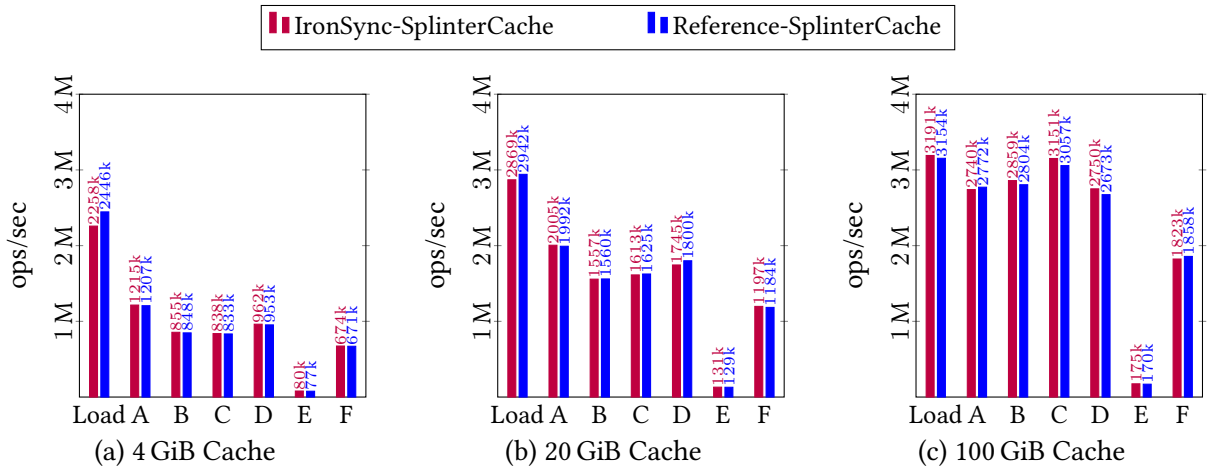
For the SplinterCache benchmarks, all of our results were run on a Dell PowerEdge R630 with a 28-core 2.00 GHz Intel Xeon E5-2660 CPU, with 192 GiB RAM and a 960 GiB Intel Optane 905p PCI Express 3.0 NVMe device.

We have two classes of benchmarks: **(i)** Our macrobenchmarks, which evaluate our verified cache in the context of SplinterDB as a whole. In these, we compare the unmodified SplinterDB to a version of SplinterDB modified to use the verified cache. **(ii)** Our microbenchmarks, which evaluate the cache on its own.

Macrobenchmarks Our benchmarks use the YCSB benchmark suite [12], a standard benchmark suite for key-value stores. We perform the standard YCSB workloads (Load and A-F) on both the unmodified and modified version of SplinterDB. Each workload uses 24 B keys, 100 B values and 14 threads. Run E performs 14M operations and the others each perform 69M operations, so that each workload logically reads/writes roughly 80 GiB of data.

We use a range of cache sizes to test different scenarios: 4 GiB to stress eviction and IO; 20 GiB to reflect a common system configuration; and 100 GiB to stress in-memory accesses and concurrency. Figure 9.23 shows that SplinterDB with IronSync-SplinterCache is always within 9% of the reference performance. Note though that 9% is the worst case, and it is usually even much better, and in some cases outperforms the original.

Microbenchmarks In our microbenchmarks, which compare the verified SplinterCache versus the original, we start by allocating and flushing pages to fill the cache. All microbenchmarks are run with a 4 GiB cache. Again, we use a spectrum of configurations that vary in the quantity



(a) 4 GiB Cache (b) 20 GiB Cache (c) 100 GiB Cache

Figure 9.23: **YCSB Benchmark for a range of cache sizes.** *Each workload is 69 M ops, except E, which is 14 M ops. Each workload uses 14 threads. Y-axis is the mean of 3 runs. Higher is better. Figure is from the IronSync evaluation [28].*

of data being accessed, which affects the pattern of eviction and the degree of thread contention. These include:

- "Uncontended" in-memory, with 2 GiB of data, (Figures 9.24a and 9.24d)
- "Contended" in-memory, with 128 KiB (32 pages) of data (Figures 9.24b and 9.24e)
- "IO bound", with 8 GiB of data (Figures 9.24c and 9.24f).

IronSync-SplinterCache is within 11% of the performance of reference on all microbenchmarks.

Node Replication

With NR, we have 3 implementations to compare: **(i)** the original NR, **(ii)** the verified NR in IronSync, and **(iii)** the verified NR in Verus. This uses the same setup as the IronSync paper [28] (which of course only compared the first two). The IronSync paper includes additional comparisons to other locking mechanisms, mostly for the purpose of demonstrating that the benchmark harness correctly configures the NUMA nodes.

Figure 9.25 shows that the three are all roughly comparable in performance, with 10% writes being the most volatile.

Mimalloc

Unfortunately, our allocator case study does not score as high on realism. Currently Verus-mimalloc only supports a subset of mimalloc's features. Our allocator *does* support:

- Allocations up to 128 KiB.
- Multi-threading.
- Overriding the system allocator on Linux (i.e., has no problematic dependencies for this purpose).

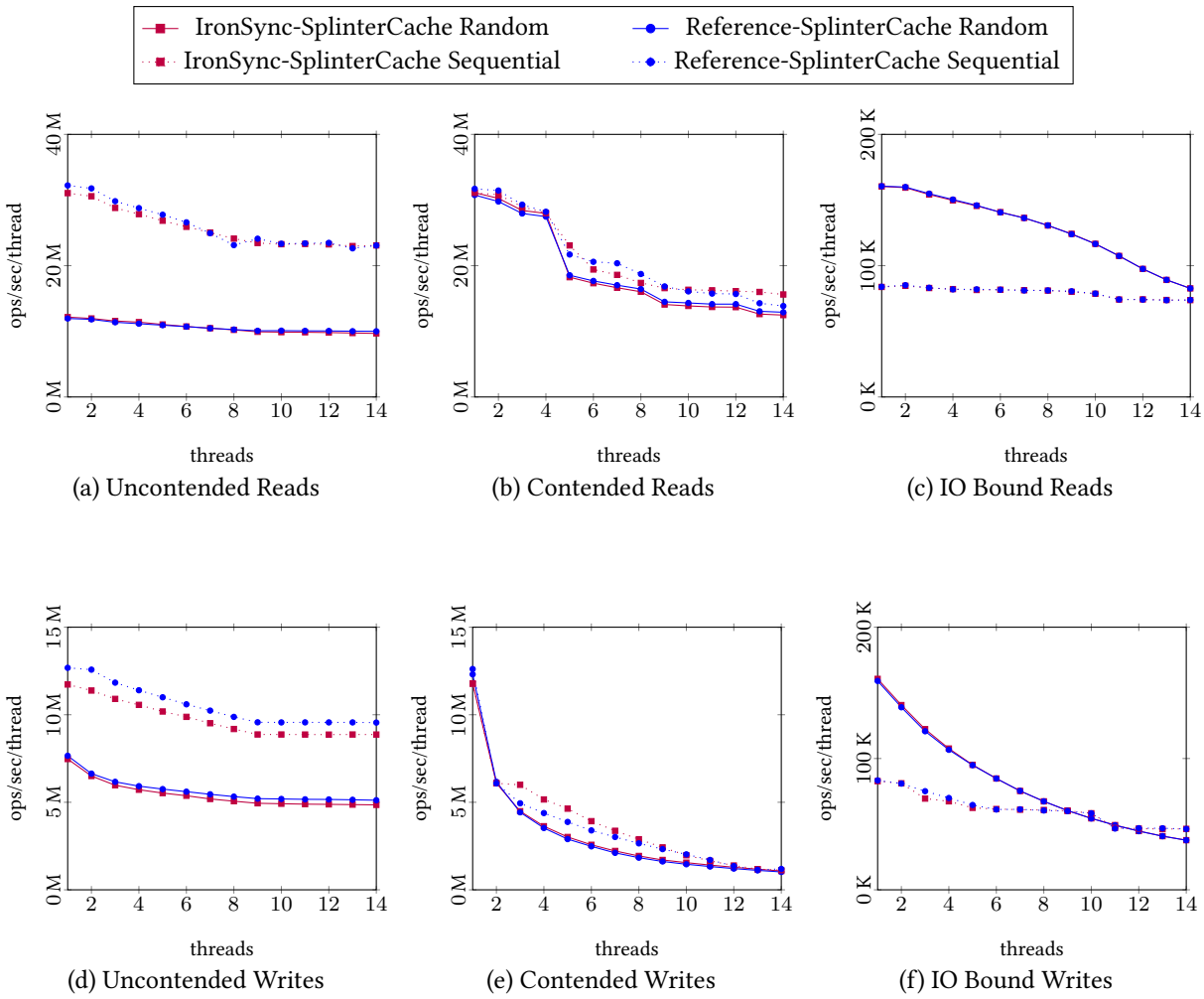


Figure 9.24: **SplinterCache microbenchmark with a 4 GiB cache.** *"Uncontended" allocates 512 MiB; "contended" allocates 128 KiB; "IO bound" allocates 2 GiB. Y-axis is mean throughput of 5 runs. Higher is better. Figure is from the IronSync evaluation [28].*

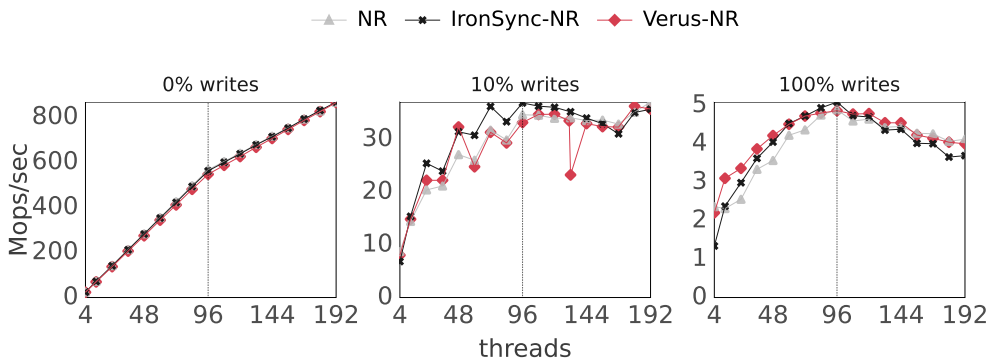


Figure 9.25: **Comparison of throughput scalability of the original NR, IronSync-NR, and Verus-NR.** *Experiments are run with 4 NUMA nodes up to 192 cores. Higher is better. Figure is from our latest Verus paper [44].*

And it does *not* support:

- Allocations larger than 128 KiB.
- Clean-up on thread termination (which is supposed to have a fairly involved process for recycling thread-local data structures).
- The `realloc` API function.
- Aligned allocations.

Because of missing functionality, we are currently able to complete 8 out of 19 benchmarks from mimalloc's benchmark suite [46] (Figure 9.26). For comparison's sake: Verus-mimalloc has about 2.7 K lines of executable code, while the original mimalloc is about 10 K lines of code.

As the benchmarks show, we also lag behind performance parity quite a bit. At this time, I do not know the primary cause(s) of the discrepancy, as it was not obvious from my initial attempts at profiling. The options range from a missing feature having significant perform impact, some misapplied decision logic (the best case, as it would be easy to fix once found), bad optimizations or instruction locality, or the use of SeqCst memory ordering instead of Release/Acquire/Relaxed (the worst case, as weaker memory orderings are entirely unsupported by our methodology).

Summary

In conclusion, in both the SplinterCache and NR case studies, we solidly met our realism goals, as demonstrated by the benchmarks. Mimalloc was a more ambitious case study, as its reference codebase is around an order of magnitude larger than either of the other two, and for mimalloc, we only had the time to implement a subset of its functionality. However, the subset we did implement is still sizable—still larger than either of the two case studies, and its algorithms and data layout are closely based on the original.

9.5.2 What did we learn?

Over the process of executing these case studies, we identified several bugs in the original implementations. It is instructive to look at how such bugs manifest in the verification setting,

Benchmark	mimalloc	Verus-mimalloc
cfrac	4.6 s.	9.7 s.
larsonN-sized	4.1 s.	12.0 s.
sh6benchN	0.14 s.	2.0 s.
xmalloc-testN	0.34 s.	0.73 s.
cache-scratch1	1.2 s.	1.2 s.
cache-scratchN	0.16 s.	0.16 s.
glibc-simple	1.2 s.	6.6 s.
glibc-thread	1.1 s.	3.6 s.

Figure 9.26: **Mimalloc Benchmarks Supported by Verus-mimalloc.** *Benchmarks run on Linux on an 8-core, 3.60 GHz Intel i9-9900K. The mimalloc authors label cfrac and larsonN-sized as "real world" benchmarks and the others as pathological stress tests.*

so I will describe each bug along with the experience of identifying it.

Bug 1: SplinterCache batch write-back bug SplinterCache has "batch write-back" functionality which works roughly as follows:

- The cache identifies a range of disk pages it wants to write back, $d, d+1, d+2, \ldots d+k$.
- For each page in this sequence, $d+i$:
  1. Look up if this page is in the cache using the *disk page map*. If in the cache, this should point to some cache entry c_i; otherwise, exit the loop.
  2. Mark c_i for write-back if possible; otherwise, exit the loop.
- Make a batch IO which writes the identified cache entries $c_0, c_1, \ldots$ to the contiguous disk range.

However, there is a critical issue here! The problem lies between (1) and (2). In step (1), we observe that c_i is mapped to $d+i$; however, this does not guarantee that these will continue to be mapped in the future. The action of setting the writeback bit *does* take the lock, but by then, it is possible that c_i maps to some *other* disk page. However, SplinterCache was not checking for this! Instead, it would always write to the originally-determined range in the last step, even if the cache entry now pointed to a different disk page.

I discovered this bug in the process of writing the verified implementation of the batch write-back routine. Specifically, I needed to establish that the cache entry had the expected disk address in order to perform the transition that initiated the IO write. Because there was no check, it was impossible to establish this, and the thus the bug became apparent.

Bug 2: SplinterCache eviction race condition We also identified a possible data race on the `disk_addr` field, which could occur when taking a lock racing with eviction.

Specifically, it is possible to have a sequence of events like this, where Thread A tries to take a shared lock on disk page d.

1. Thread A tries to take a take a lock on disk page d. It checks the disk page map and finds that it is in cache entry c.

2. Thread A increments a read-counter for cache entry c.
3. Thread A checks that the exclusive lock is not already held for c.
4. Thread B evicts the cache entry c.
5. Thread C loads a different page into the cache entry, modifying the `disk_addr` field.
6. Thread A reads the `disk_addr` to check if it is still equal to d.
7. Thread A checks that the entry is not in the *loading* state.

Observe that (5) and (6) could happen in either order as there is nothing synchronizing them, resulting in a read-write data race.

I encountered this issue while trying to read `disk_addr` field in step (6). To understand the issue, first note that I had modeled the process of taking the shared lock on c as consisting of 3 steps: the ones numbered (2), (3), and (7). I knew that after performing these three steps, I would have safe access to the `disk_addr` field, but I got stuck when I realized I needed to read the `disk_addr` field *before* step (7). After noticing this, I was able to construct the problematic interleaving.

To fix this bug, we can simply move the `disk_addr` read (step (6)) to be after the loading state check (step (7)). This is proved safe under the locking scheme, and the result is that atomic change to the loading state synchronizes the write and the read.

Bug 3: Node Replication linearizability violation NR is expected to provide a linearizability guarantee, but there was originally a bug in this guarantee.

Consider two three threads: T_1 and T_2 on one node $node_T$, and thread S on a different node, $node_S$.

Now consider the following sequence of events:

1. The replicas at $node_S$ and $node_T$ are both at version v; `version_upper_bound` is also at v.
2. Thread T_1 becomes the combiner thread for $node_T$.
3. Thread T_1 executes an operation from the message buffer, updating $node_T$'s replica from version v to $v + 1$.
4. Thread T_2 performs a query against version $v + 1$ of $node_T$.
5. Thread S performs a query, against version v of $node_S$.
6. Thread T_1 reaches the end of the combiner phase and updates `version_upper_bound` to $v + 1$.

This sequence is clearly problematic: Step (4) can complete before Step (5) begins, resulting in a linearizability violation because Step (4) returns a result from a later version than Step (5) does.

First of all: How is NR *supposed* to prevent this problematic interleaving from happening? When performing a query, a thread is supposed to wait until its local replica reaches the version indicated in `version_upper_bound`. However, this doesn't seem to help: after all, this rule is perfectly consistent with the above interleaving because thread T_1 does not update `version_upper_bound` to $v + 1$ until the end.

The actual problem with this interleaving, at least according to the intended design, is that T_2

finishes its query *too early*. If T_2 were forced to finish its query later, *after* `version_upper_bound` gets updated, then there would be no linearizability issue: S would still observe a version earlier than what T_2 observes, but T_2 finishes later so there is no ordering problem. And in fact, there *is* a mechanism that prevents T_2 from finishing early. Namely, T_1 holds the replica lock for the duration of the combiner phase, so T_2 cannot access the replica until it completes. Thus, the problematic interleaving is ruled out.

However, it turned out that this reasoning was only true in the common case. There was a less common case which only triggered when the log became full, and in that case, the combiner phase ran with a different locking discipline. Specifically, the combiner would take the replica lock only around each individual replica update, rather than around the entire combiner phase. As a result, the above interleaving became possible.

We identified this bug fairly early in the verification process while we were designing the UnboundedLog(**X**) system and trying to reason out the linearizability argument. After identifying the problematic interleaving, the NR developers were able to explicitly reproduce the issue in NR.

For our verified NR, we had to add a precondition in UnboundedLog(**X**) that a query can only take place when the `combiner` token is in the `Idle` state. This precondition was ultimately crucial in the linearizability argument. Furthermore, it meant that we had to put the `combiner` token behind the replica lock, which ultimately meant that we could only implement the combiner phase by holding onto the replica lock throughout. This is why it was impossible for the verified version to make the same implementation mistake.

Bug 4: Mimalloc race condition for multi-threaded free We found a data race in mimalloc related to multi-threaded free and *aligned* allocations. An aligned allocation is when the user requests that the allocation address be a multiple of some given integer. To satisfy the request, the allocator will first allocate a block as usual, then return a pointer which is possibly offset from the boundary of the block.

When performing a `free`, the code has to find the boundary of the block, so to do this, it read some state related to aligned allocations. However, this turns out to be a data race for multi-threaded free, since that same data could be written to while concurrently performing an aligned allocation.

I noticed this bug, even though I have not added support for aligned allocations, because I was unable to access the relevant state while performing the thread-local free. This bug was also discovered and reported by an independent party around the same time I discovered it.[5]

9.5.3 How much effort was it?

There are many dimensions of proof effort. There is the human effort to write the proofs, and there is the CPU time needed to verify the proofs. Admittedly, the latter is mostly important only in so far as it impacts the former, but it *does* impact it. Long verification times result in protracted feedback cycles and developer frustration, so it is highly desirable to keep them small.

[5] https://github.com/microsoft/mimalloc/issues/865

Development time

Technically, the verified SplinterCache took around 18 months to develop, from the time we began studying it; however, this includes the time it took to design and implement the IronSync framework itself, so it is hard to say how much time SplinterCache actually took "on its own." When we started NR, the IronSync framework was much further in development. It took around 6 months. Porting NR to Verus naturally was a much quicker endeavor, since most of the proof work carried over cleanly.

The mimalloc case study took around 9 months, though again, much of the time spent here was necessary work on Verus itself.

Line counts

In Table 9.1, we categorize source lines into three buckets: Trusted (which includes specifications and trusted interfaces), Proof, and Implementation. We compute the "proof-to-code" ratio, giving a rough measure "amount of proof effort per line of implementation."

Node Replication has the highest proof-to-code-ratio. It actually has around as much proof code as SplinterCache, but a much smaller implementation. We also see the amount of proof code decreases a bit in the port from IronSync/Linear Dafny to Verus. The proof-to-code ratio went down by quite a bit, though this is partially because the implementation lines increased.

Comparison to the state-of-the-art It is difficult to make sound, "apples-to-apples" comparisons on proof effort due to the variety of systems of interest, all with different properties, different proof guarantees of interest, and different proof tools where "a single line of proof code" might not represent the same amount of "effort." Nonetheless, a quick survey of systems verification projects illustrates that the proof effort is often substantial:

- The seL4 microkernel (2009) [38] reports 200 K lines of proof for 8.7 K lines of C code. (Proof-to-code ratio of 23). This does not include multi-core support.
- IronFleet (2015) [29] reports 39 K lines of proof for 5.1 K lines of Dafny code. (Proof-to-code ratio of 7.6). They handle distributed systems but not shared-memory concurrency. They also handle liveness.
- CertiKOS (2016) [24] reports 90 K lines of proof for 6.5 K lines of code. (Proof-to-code ratio of 13.8). They handle multi-core with fine-grained locking and also show termination.
- VeriBetrKV (2020) [25] (my own work) reports 47 K lines of proof for 6.4 K lines of Dafny code. (Proof-to-code ratio of 7). VeriBetrKV handles crash safety but not shared-memory concurrency.[6]
- GoJournal (2021) [9] reports 25.7 K lines of proof for 1.3 K lines of Go code (Proof-to-code ratio of 19). GoJournal performs concurrent reasoning using Iris.

[6]These numbers are for the first version of VeriBetrKV, which used Linear Dafny, but only minimally. Li et al. [51] report that when translating nearly all of VeriBetrKV to Linear Dafny, the lines of proof in VeriBetrKV's implementation layer reduce by 28%. The implementation layer is itself around 55% of VeriBetrKV; this means that the proof-to-code ratio of VeriBetrKV as a whole drops from 7 to approximately 6.

Of these, the most comparable in terms of *scope* is probably GoJournal. It is of similar scale to our major case studies, requires reasoning about a storage system, and is concurrent. Even still, there are major differences: GoJournal proves more substantial crash-safety properties, and it has the benefit of being foundational.

Setting aside these caveats on the difficulty of comparison, we can conclude that our proof-to-code ratios are fairly favorable.

Verification times

Table 9.1 also reports verification times. The times are measured with 1 CPU, so it should be noted that our method of verification is highly parallelizable, and thus it is even faster using all the cores on a machine.

Again, Node Replication forms a point of comparison between IronSync/Linear Dafny and Verus. Surprisingly, the verification time for Node Replication is improved by *two orders of magnitude*, rendering it basically negligible. Furthermore, the verification time for the memory allocator, despite it being the largest case study, is shorter in Verus than either of the Linear Dafny case studies.

Comparison to the state-of-the-art Historically, verification times have been a massive pain point for SMT-based verification. IronFleet [29] reports 6.5 hours of CPU time for 39 K lines of proof, VeriBetrKV [25] reports 1.8 hours of CPU time for 44.6 K lines of proof and Armada [53], a highly automated framework for low-level concurrency verification, takes 4.9 hours of CPU time to verify 70 lines of code.

IronSync is comparable to VeriBetrKV in time-per-proof code despite supporting shared memory concurrency. Furthermore, Verus's verification times are a significant improvement beyond that.

Verus compares favorably even to some frameworks from the interactive theorem proving world. For example, RefinedRust [21] reports 22 minutes to verify `Vec`. However, this comparison is somewhat unfair; RefinedRust and other work that use Coq/Iris aim at being *foundational*—i.e., having a minimal TCB—which means automation take a lot more work. Any optimizations need to be proved correct, not just on paper, but mechanically. There is no doubt Verus would take much longer if it was also generating and checking Coq proof terms.

Why is Verus faster than Linear Dafny? There are several factors contributing to Verus's speed. Determining the most significant factors scientifically would require a number of time-consuming experiments that isolate each factor independently. However, I believe it is nearly certain the primary factor is Verus's lean verification condition generation, developed by Chris Hawblitzel.

For comparison's sake, Linear Dafny generates 2063 MB of SMT-LIB (the format used for verification conditions that Z3 takes as input) for the NR case study. Verus generates only 22 MB of SMT-LIB for the same case study. The orders-of-magnitude reduction roughly corresponds to the orders-of-magnitude reduction in verification time. Our most recent Verus paper [44] performs a similar comparison for a different case study: IronFleet's IronKV [29], ported from

	Trusted	Proof	Impl	Proof-to-code ratio	Verif. time
SplinterCache (IronSync)	771	8670	1579	5.5	896 s.
Node Replication (IronSync)	104	7828	730	10.7	1089 s.
Node Replication (Verus)	469	6678	894	7.5	17 s.
Memory allocator (Verus)	282	13241	2717	4.9	262 s.

Table 9.1: **Case study LoC totals, proof-to-code ratios, and verification times.**

Dafny to Verus. IronKV goes from 352 MB and 445 seconds to verify to 41 MB and 41 seconds to verify.

But why *does* Verus have such leaner verification conditions? One factor is its aggressive pruning of irrelevant dependencies. Another factor is its insistence on making spec functions *total*, which eliminates the need for checking preconditions of spec functions.

A secondary factor for Verus's verification times (which would not be accounted for in the SMT-LIB size reduction) is its more conservative quantifier trigger strategy (§3.3.3). This more conservative strategy results in smaller proof spaces and likely has an effect in reducing outlier verification times.

Line count breakdowns and individual VerusSync systems

Table 9.2 subdivides the Proof code in order to get more insight into the various uses of VerusSync and IronSync's monoid interface. The contents of the "Other" category is mostly proof code intertwined with the implementation.

NR, having been fully ported to Verus, provides a solid point of comparison between IronSync's monoidal transition systems and VerusSync. The main goal of VerusSync was to reduce boilerplate code, and this is validated by the drastically reduced line counts; every VerusSync system in NR has been reduced by 40% or more compared to the IronSync counterpart. We discussed factors contributing to the code reduction in §8.1.

Of all the transition systems across the board, NR's UnboundedLog has the highest line counts in both IronSync and Verus. This is pretty consistent with my subjective recollection of the development—UnboundedLog had some fairly involved invariants to be proved.

Interestingly, the most complicated part of the memory allocator proof turned out to not be a VerusSync component at all. Rather, that distinction goes to a module containing the invariant proofs regarding segment layout and the doubly-linked lists of pages. This aspect was entirely thread-local, so I did not attempt to use VerusSync, though it might be interesting to investigate if a more separation logic-style approach could have helped anyway.

9.6 Addressing the challenges

In this section, we will review how we were able to address the challenges identified in Chapter 2, and I offer my commentary.

SplinterCache (IronSync)	LoC	
Spec	185	TRUSTED
Disk Environment Model	586	TRUSTED
Cache (monoid)	1036	PROOF
CacheRwLock (monoid)	2015	PROOF
Refinement Proofs	2456	PROOF
Other	3163	PROOF
Impl	1579	IMPL
Node Replication (IronSync)	LoC	
Spec	104	TRUSTED
UnboundedLog (monoid)	2329	PROOF
FlatCombine (monoid)	649	PROOF
CyclicBuffer (monoid)	1756	PROOF
NRRwLock (monoid)	633	PROOF
Refinement Proofs	1291	PROOF
Other	1170	PROOF
Impl	730	IMPL
Node Replication (Verus)	LoC	
Spec	469	TRUSTED
UnboundedLog (VerusSync)	1445	PROOF
FlatCombine (VerusSync)	248	PROOF
CyclicBuffer (VerusSync)	573	PROOF
NRRwLock (VerusSync)	224	PROOF
Refinement Proofs	1288	PROOF
Other	2900	PROOF
Impl	894	IMPL
Memory allocator (Verus)	LoC	
Spec	37	TRUSTED
Thread IDs & mmap	245	TRUSTED
Mim (VerusSync)	1066	PROOF
Segment slices	4177	PROOF
Other	7998	PROOF
Impl	2717	IMPL

Table 9.2: **Line count breakdown for our major case studies.**

Challenge SpC-2 (Specialized Lock). We need to be able to reason about specialized lock implementations that support read-locks and write-locks. Read-locks may be taken simultaneously by multiple threads, while write-locks must be unique.

Challenge NR-1 (Specialized lock-like system). We need to be able to reason about specialized implementations that support simultaneous read-states and exclusive write-states, *including those that do not resemble traditional reader-writer locks.*

Challenge Mem-2 (Multi-threaded free). The correctness of `free` is reliant on the client calling it correctly, and from this information, we need to be able to make a number of deductions about the behavior of a multi-threaded system.

We solved these locking problems using the storage protocols and the deposit/withdraw/-guard pattern, though at this point, I have little to say on the pattern that I haven't said already.

I will only remark on one specific commonality. We motivated the deposit/withdraw/guard pattern by observing the lifetime-bounded functions in Rust's `RefCell` and `RwLock` interfaces. However, not one of these three systems had any objects whose role resembled that of `Ref` or `RwLockReadGuard`. This is part of why it was so important that we could guard on *ghost* shared references: There was nothing else to guard on! Thus we had to introduce our own objects to play the role.

Challenge SpC-1 (External devices). We need to be able to reason about the properties of a system where the program is but one component interacting with external devices.

We solved this using the GSM method. The GSM method let us abstract the global operation of the program as a state machine, and then, using fairly classic state machine-style reasoning, we were able to reason about the operation of the program when connected to a disk with certain assumptions.

Challenge NR-3 (Future-dependent linearization points). We need to be able to prove linearizability even in the presence of future-dependent linearization points.

Reasoning about future-dependent linearization points was challenging because Verus does not have any "prophecy variable primitive" or any other primitive giving future insight. The GSM method allowed us to approach the problem by reasoning about traces, though trace-based reasoning has the danger of being quite unwieldy. To avoid this, we factored the refinement problem into steps. The first step, a state-based refinement, allowed us to abstract the system state down to something much more manageable, where it became practical to do the requisite trace-based reasoning.

In terms of concrete output, our refinement stack was one of the most successful elements of the NR project, as it led us to identify an actual bug in the original NR. Despite this, the GSM method, as-is, is somewhat unsatisfactory on a purely technical level, since it doesn't help us obtain a usable Hoare-style specification for the system, making it difficult or impossible to compose vertically and verify the client of NR.

One possible direction for addressing this would be to integrate *prophecy variables*, following in the footsteps of work on using prophecy variables in separation logic [37] to resolve future-dependent linearization points. Unfortunately, prophecy variables are a little tricky to implement soundly in Verus. If designed naively, it would be easy to accidentally allow some kind of "time travel grandfather paradox":

```
proof fn grandfather_paradox() {
    let tracked proph = Prophecy::<bool>::new();
    let x = proph.value();    // (1)
    let y = !x;
    proph.resolve(y);         // (2)
    assert(x == proph.value());     // By line (1)
    assert(y == proph.value());     // By line (2)
    assert(false);
}
```

Thus, implementing prophecy variables in Verus requires careful consideration, and it is difficult to do it flexibly enough to support all the target use-cases.

The ideal scenario, to me, would be to determine a way to soundly support prophecy variables in Verus and show that the trace refinement method is generically implementable by prophecy variables. However, there are some obstacles to doing so. One scenario we have to consider is the possibility that the data structure **X** is itself dependent on ghost state. In that scenario, attempting to prophecize the future values of the UnboundedLog(**X**) state would result in the prophecization of ghost state values, and this is exactly the kind of thing that leads to potential unsoundness. I leave this as a challenge for future work.

Challenge SpC-3 (Intertwining). Logic related to high-level cache domain logic is intertwined with low-level synchronization logic, which increases the complexity of the implementation.

Challenge NR-2 (Intertwining). Logic related to high-level replication and linearizability domain logic is intertwined with low-level synchronization logic, which increases the complexity of the implementation.

Both SplinterCache and NR had a similar approach to this. Each one used a VerusSync system (or the monoidal IronSync equivalent) to reason about the "high-level domain logic" and each one used a different VerusSync system (or the monoidal IronSync equivalent) to reason about the "low-level synchronization logic."

Interestingly, the two systems have overlapping state. At a purely *technical* level, this might not seem so interesting—the ability to maintain an invariant between a single atomic word of memory and two different ghost state systems was hardly worth mentioning in the formalization.

What always intrigues me about this fact is the impact on the high-level proof architecture. As a *software engineer*, I usually think of "modularity" in terms of libraries, when you have some complex implementation abstracted away by a method call with some reusable specification that enables easy reasoning. In the case of SplinterCache or NR, though, we abstract the code down in a way that doesn't cleanly line up with method call boundaries. Even so, the resulting abstraction has its own self-contained logic that doesn't 'leak out.'

One objection might be that we still have to intermingle the two together in a complex implementation that then entangles everything together. While this is certainly cumbersome, in practice it seems to be *conceptually* simple once the relevant systems are identified and factored out.

Challenge Mem-1 (Fungible memory). We need to manually organize the address space, and safely divide the memory between internally-used memory and memory provided to the client used incorrectly.

Of course, we solved this one with memory permissions.

One point I have not remarked much on is the process of actually *meeting the preconditions* of all the pointer operations. The conditional safety of a pointer operation dictates that we always prove the `PointsTo` permission actually corresponds to the address that we are accessing. Usually, this is pretty straightforward compared to actually getting ownership of the `PointsTo` in the first place.

In the case of the memory allocator, however, this frequently required a lot of arithmetic proofs. In my prior experience, proofs involving nonlinear arithmetic tended to be somewhat taxing in SMT-based program verification. I believe Verus isolating nonlinear arithmetic (as described in §3.3.3) helped avoid a lot of frustration, but it still involved more manual effort than I would have liked.

Challenge RC-1 (Simple spec). The formal specification of a smart pointer should be easy and convenient to use by the client, matching the informal reasoning that a pointer-handle to a `T` is "like" a `T`.

This one was pretty straightforward in the end. I'll only remark that this goal was achievable primarily because Rust's `Rc<T>` and `Arc<T>` types only allow you access to a shared immutable reference, `&T`. If we were dealing with shared pointers to some mutable data structure (like C++'s `shared_ptr`, for example), the situation might have been more complicated.

Challenge RC-2 (Thread (non-)safety). Our framework should be able to handle objects that are *not* thread-safe and the more permissive implementations they permit, while still ensuring that they are not used incorrectly.

Rust's `Send` and `Sync` marker traits helped us solve this one. Our contribution was to have ghost tokens that operate smoothly within these traits.

Challenge RC-3 (Recursive types). Our method needs to be consistent with the use-case of recursive data structures.

The Verus implementation doesn't have to do anything special to make recursive uses of `Rc` or `Arc` work. However, a substantial amount of the design and theory had to be crafted to support this use-case. For one thing, the `LocalInvariant` and `AtomicInvariant` types need to work with recursive types. This is possible because of the trait-based mechanism for specifying the invariant predicate as we discussed in §6.3.6. With a more naive method, `LocalInvariant<V>` would count as a "negative position" of `V` for the purpose of Verus's type consistency checks.

There is another subtle issue as well. `VerusRc` contains the field:

`RefCounterTokens::reader<MemPerms<S>>`

which is a ghost token from a storage protocol. Therefore, storage protocol tokens need to be allowed in recursive types as well. In λ_{Verus}, it was only possible to support recursive types simultaneously with storage protocols because of Leaf's nontrivial SP-Exchange-Guarded rule and its variants. Without these, we would not have been able to prove sound operations like `exchange_nondeterministic_with_shared` in the way we did. Instead, we would have been forced to prove mask disjointness, and that would have required some kind of stratification of masks that would have been inconsistent with recursive types.

Chapter 10

Related Work

10.1 Linear Types and Ownership Types

The idea of using a type system to manage resource ownership goes all the way back to Girard's linear logic [22], its application to computation [23], and Wadler's linear types [78]. Since then, linear types have seen use in systems verification in order to make it easier to reason about mutable-in-place data structures in a more functional, less imperative manner.

For example, Cogent [1] is a functional language that uses a linear type system to allow updates-in-place. The authors show how Cogent can be used to write a verified file system. In fact, Cogent was one of the inspirations for Linear Dafny. Cogent also uses its linear type system to catch memory leaks. On the other hand, neither Verus nor (despite its name) Linear Dafny have a guarantee of leak-freedom.

Linear Dafny, as we have discussed, was a direct precursor to Verus. One feature of Linear Dafny that I have not remarked upon is its relationship to Dafny's traditional means of working with mutable state, that is, its *dynamic frames*. Linear Dafny supports both linear types and dynamic frames, and it also supports a mechanism for "encapsulating" dynamic-frame-based reasoning into a linearly-manipulated object called *regions* [50]. This is not a feature that has made it into Verus, which (following in the Rust lifestyle) goes all in on ownership; I did not make use of regions in the IronSync work, either.

In principle, it should be possible to emulate regions using ghost permission objects, but ghost permission objects provide some more flexibility. For example, a verified doubly-linked list in Linear Dafny, even with regions, will still have to constantly reason about all the nodes being at distinct heap locations. The Verus version of the doubly-linked list (§3.5.1) does not have to do this. That said, there are perhaps applications where the 'regions' approach would be more lightweight than the 'explicit ghost state' approach.

10.1.1 Permissions through substructural ghost types

To my knowledge, the first language to treat permissions as *values* was $\mathbf{L}^3$ [58]. $\mathbf{L}^3$ had a non-linear pointer type Ptr ρ and a linear permission type Cap $\rho\ \tau$, thus tying the permission to the pointer via the "location variable" ρ. The Rust GhostCell work [79] is similar in this regard, albeit for cells with interior mutability rather than pointers. Specifically, GhostCell uses

a "brand parameter" `'id` to tie together a cell type `GhostCell<'id, T>` with its permission type `GhostToken<'id>`. By contrast, in Verus, the connection between the pointers/cells and their permissions are not in their types but in their *specs*. Practically speaking, that means that users have the full power of general-purpose theorem proving to show that the pointers or cell IDs line up when performing such an operation.

One other technical difference between `GhostCell` and our `PCell` is that accessing the interior of a `GhostCell` *does not require the permission if the* `GhostCell` *is exclusively owned* (as opposed to shared via a reference). By contrast, `PCell` *always* requires the permission (e.g., see `into_inner` in Figure 3.2). The reason we need this restriction is that `cell::PointsTo<T>` gives access to all the same ghost invariants at a ghost `T` would, and for this to be sound, we need to be sure that the `PCell<T>` cannot do the same. At this time, I have not characterized exactly what, if anything, is lost by not giving these extra powers to exclusively owned objects, so it is left for future work to determine if this is a significant gap in Verus. It is also not clear if our different approach to the semantic interpretations of types (as compared to RustBelt) would support this property of `GhostCell`.

The Linear Maps work [40] is another work that uses linearly tracked ghost heap objects. This work is like ours in that it targets verification, uses specifications encoded in first-order logic to be discharged by SMT solvers, and positions itself as a way of bring separation logic strategies into that setting.

A common limitation in the earlier work such as $\mathbf{L}^3$ or Linear Maps is the lack of substantial support for read-only, temporarily shared permissions. Verus can support shared permissions thanks to Rust's complex lifetime system and because of our storage protocol rules. GhostCell also supports shared references to ghost tokens via Rust's lifetime system.

10.2 Separation Logic

We have seen throughout this thesis that our techniques are closely related to separation logic. Verus primitives are directly inspired by specific techniques from separation logics, and we discussed formal connections in Chapter 4 and Chapter 6. But how does our method compare to state-of-the-art separation logic in the actual *practice* of verifying code?

10.2.1 Verus specs versus separation logic specs

In separation logic frameworks, "ghost state" is usually not part of the source code in the same way that it is for Verus.[1] In separation logic frameworks it usually works like this: You have some program (with normal, executable code instructions), and then to the side, you have some proofs about the program behavior. The "ghost state" is entirely within the resource logic used by these proofs.

There are advantages and disadvantages. One advantage is that separation logic specs usually look a lot *cleaner*. For one thing, the proof is not mixed in with the code the way it is in a tool like Verus. Furthermore, separation logic has a unified way to reason about both resources and "pure mathematical facts" as propositions. For example, you can have a points-to proposition

[1] Though there are some exceptions to this, like the ghost code used to handle prophecy variables [37].

$\ell \hookrightarrow v$ and a fact about v like $v \neq 5$ and carry them around together as $(\ell \hookrightarrow v) * (v \neq 5)$. Meanwhile, if you wanted to do something like this in Verus, you would:

- Add a "tracked" ghost variable `points_to` that corresponds to the points-to proposition.
- Add some mathematical specification: you'd write that `points_to.loc()` is equal to ℓ, that `points_to.value()` is equal to v, that $v \neq 5$, and so on.

This is to say nothing of other connectives like the magic wand $(-\!\!*)$ or the wand update $\Rrightarrow\!\!\!*$, which are cumbersome to emulate in Verus.

On the other hand, there are some advantages to having this ghost state as part of the source code. By being part of the source code, these ghost objects get to interact with the type system. We can track ownership through the lifetime system, marker traits are able to propagate through the ghost objects, and so on.

10.2.2 Verus specs versus implicit dynamic frames

Implicit dynamic frames [65, 68] is a relative of separation logic that is also based on permissions, but the specifications are written in a slightly different way. Rather than a $\ell \hookrightarrow v$, you just have a permission for ℓ, or usually a field like $\ell.f$, written $\mathrm{acc}(\ell.f)$. Then in specifications, you dereference $\ell.f$ directly, which is only well-formed when you have the appropriate permission. Verification tools that incorporate implicit dynamic frames, like Chalice [49] or Viper [60], usually include fractional permissions as well.

10.2.3 Automation

On the whole, our methodology is a high-automation one, using SMT solvers to dispatch most proof obligations, including postconditions on executable functions, the proof obligations demanded by VerusSync, and miscellaneous lemmas. However, when we zoom in on the *ownership discipline in particular*, we find one aspect that is highly manual, namely the manipulation of ghost objects. The ownership type system (Rust's or Linear Dafny's) is fast and efficient, but this is arguably enabled only because the user has to manually orchestrate the ghost state in the source code. For example, if the developer reads from or writes to a pointer, they have to supply the exact ghost permission object that justifies that access. It is never inferred from ambient context.

How does this compare to separation logic work?

Iris Iris proofs are usually written in Iris Proof Mode (IPM) [39], a Coq tactics framework for manipulating spatial hypotheses. Because the means of interaction is so different, it is difficult to make a direct comparison, but IPM is similar to our methodology in that it usually requires the user to manually direct which hypotheses to which conclusions. For example, given a goal like $\mathit{Hypotheses} \vdash A * B$, the user (in all but the most trivial cases) has to manually choose which hypotheses will be used to prove A or B. However, some work has extended Iris with additional automation, such as Diaframe [59], which uses a hint system to make progress on goals like $\mathit{Hypotheses} \vdash A * B$.

Steel Steel [20] is a separation logic framework in F* which uses SMT to solve framing. Steel is also notable for offering CSL proofs for a dependently-typed language. Their paper positions Steel as aiming to "provide pragmatic automation for simpler code through dependently typed proof-oriented libraries" rather than "tricky concurrent algorithms."

Viper Viper [60] is another separation logic framework using SMT. In Viper, the programmer is expected to provide manual "fold" and "unfold" annotations for permission definitions. (In Verus, the closest equivalent would probably be something like "wrapping or unwrapping a ghost object from inside a struct," which also needs to be executed manually.) Otherwise, Viper automates framing, and it also supports the magic wand operator.

10.2.4 Ghost State Construction Mechanisms

Nearly all discussion of automation in CSL pertains to framing, and there is much less on automating ghost state construction in the vein of VerusSync. I'll briefly discuss some other paradigms for ghost state construction.

Combinators The traditional Iris approach to constructing ghost state is to use a library of *Resource Algebra combinators*, i.e., type constructors that build RAs out of smaller RAs or other parameters. In principle, we could build a similar library using the traits in the Verus Monoidal Ghost Interface interface, though this is not the direction that this thesis prescribes. Partly this is because I am not aware of any useful combinators for storage protocols.

VerusSync could, perhaps, be viewed as a particularly complicated combinator; after all, in §5.5, we saw how to formalize VerusSync Core in terms of a product RA construction out of several commonly used RA. The one aspect of VerusSync Core that is not easy to replicate purely with standard RA combinators is the invariant predicate. In idiomatic Iris, the invariant predicate would usually go in, well, an invariant.

STSs One of the oldest Iris constructions is the *State Transititon System (STS)*, defined by CaReSL [75] and encoded into PCMs in Iris 1.0 [31]. An STS describes a set of states, transitions between the states, and mappings between states and tokens. On the surface, at STS sounds a lot like VerusSync; however, STSs have mostly fallen out of favor, with the Iris codebase now warning that "STSs are very painful to use in Coq, and they are therefore barely used in practice" [72]. The reader might ask, what is different about VerusSync that makes it usable, and why is it not merely retreading old ground?

One major difference is that, whereas as STS requires the user to explicitly define the token set, this is an automated procedure in VerusSync. Furthermore, because of the split between our sharded interpretation and unsharded interpretation, we are able to generate all proof obligations in the unsharded interpretation, i.e., without having to reason about tokens at all. Furthermore, this design was explicitly inspired by prior work for generating efficient verification conditions, and it takes advantage of high-automation SMT solvers.

10.2.5 Shared, read-only state

In separation logics, the most common way to handle read-only shared state is via *fractional permissions*. In Iris, for example, it is customary to make a fractional points-to proposition $\ell \overset{\text{frac}}{\hookrightarrow}_q v$, and likewise in Viper [60], all memory permissions are fractionalized.

Fractional permissions are based on a fairly elegant idea: if a permissions are treated as fungible *quantities* summing up to 1, then you can treat 1 as a write permission and smaller nonzero fractions as read permissions. This is sound because whenever someone has the permissions of quantity 1, it is guaranteed there aren't any readers anywhere else, though they are free to write without interfering with anybody else.

Despite the elegance of this system, fractional permissions rarely correspond to the way developers think about programs, so implementing nontrivial read-sharing patterns incurs a lot of mental overhead in trying to map fractions to a given problem domain.

In Verus By contrast, shared references (like in Rust) have proved to be a fairly intuitive way to think about shared state, so we take advantage of this in Verus. The storage protocol's guarding system allows us to express nontrivial sharing protocols using shared references and bounded lifetimes. Thus, we avoid fractional reasoning entirely. (In principle, fractional permissions can be implemented via a storage protocol, but in practice, I have not found the need to come up.)

In Iris Setting aside Verus for a moment, Leaf (§4.5) also provides an alternative to fractional permissions in *Iris*. We demonstrated this in a number of ways in Chapter 6: We used points-to propositions without fractions; we made "cancellable invariants" without fractions, and we handled shared references using a lifetime logic without "fractional borrows."

Our lifetime logic did not employ fractional lifetime tokens, either, although it is worth noting that some of RustBelt's more interesting uses of fractional lifetime tokens are relevant only when its "full borrows" are in play (and they are not in play for the Leaf Lifetime Logic we presented). For example, RustBelt's LftL-Bor-Acc rule [33] requires the user to temporarily give up a fractional lifetime token:

$$\&^{\kappa}_{\text{full}} P * [\kappa]_q \Rrightarrow \triangleright P * (\triangleright P \Rrightarrow \&^{\kappa}_{\text{full}} P * [\kappa]_q)$$

Thus, for example, writing down the "Leaf analogue" of this rule and proving it remains future work, In the future, I hope to continue exploring and testing the limits of Leaf's expressivity.

10.2.6 Handling future-dependence

We discussed future-dependent linearization points while handling NR (§9.2). Future-dependent linearization points have also been handled in the Iris separation logic using *prophecy variables* [37], a method which is quite a bit different than the one we used.

The key idea we used for NR was to handle linearizability "outside" the program logic as a standalone theorem. As a result, it is difficult or impossible to compose vertically and verify the client of NR, or at least it is if we want to use the obvious linearizability-based spec we worked so hard to prove. On the other hand, the prophecy variable approach allows the developer to reason directly about future values in the program logic.

The Iris work uses a ghost resource to represent the right to resolve the prophecy, preventing contradictory resolutions. They use an explicit notion of "ghost code," distinct from ghost state in the resource logic, so it not possible for ghost state (which might depend on prophecized values) to influence the results of a prophecy. They also show how how to handle future-dependent linearization points using this method, and their specification expresses the notion of linearizability using the Iris concept of *logical atomicity*.

It would be interesting to integrate prophecy variables into Verus and try to combine the approaches; we discussed this earlier in §9.6.

10.2.7 Refinement

Our GSM method is based on the idea that we can prove some specifications about ghost resources in order to establish a refinement between a program and a more abstract spec. Using ghost resources to relate Hoare triples and refinement goes back to CaReSL [75].

Perennial [8] also follows the CaReSL approach, but develops their refinement theorem in the crash-recovery setting. Their resources and specification triples were a direct inspiration for the request/response system in our GSM concept. Specifically, Perennial describes their main refinement theorem in terms of three kinds of resources: one representing the abstract state, one representing an operation to be performed, and one representing a return value. Our GSM method is similar, except rather than one resource to represent the abstract state, we emphasize that this state too should be thought of as a composition of resources, and that transitions should be viewed as local operations. We compare further to Perennial in §10.4.

Trillium [73] is an Iris separation logic framework that establishes refinements between programs and labeled transition systems. Thus, they are able to show *foundationally* that an implementation is a refinement of an abstract TLA$^+$ specification. They are also able to handle liveness properties.

10.3 Rust Verification

Today, there is a broad spectrum of Rust verification tools. **Prusti** [2] and **Creusot** [18] are two tools similar to Verus in that they both allow annotating functions with preconditions and postconditions. Prusti uses a separation logic engine called Viper [60] as its backend, while Creusot uses a first-order logic encoding in Why3 [4]. Creusot is notable for implementing the *RustHorn encoding* [55], in which mutable references are encoded via prophecy variables, and the need for separation logic reasoning is avoided entirely. Among the sea of Rust verification tools, Creusot is likely the one whose encoding is most similar to Verus, though Verus does not (yet) have general mutable reference support.

To my knowledge, neither Creusot nor Prusti has explored the use of ghost state to tackle unsafe code to the extent we have with Verus, though I do not know of any inherent limitation that would prevent it.

Aeneas [30] is a tool with a substantially different architecture to Verus: its proofs are *extrinsic*, i.e., they are not intermixed with executable code in the source files, but they exist "to the side." This is done by translating the Rust functions to models of the functions expressed

in a *functional* style. Proofs about these functions are then written in an interactive theorem prover, e.g., Lean [15]. Rather than a prophecy-encoding, it represents mutable references by its concept of "backwards functions." The idea behind a backwards function is that when a function takes a mutable reference and returns a mutable reference, they encode this as a function which returns a function that computes the new value of the input mutable reference given the final value of the output reference. Another notable feature of Aeneas is that it does not need to trust the results of rustc's lifetime-checking, instead relying on its *Low-Level Borrow Calculus* to capture borrow semantics.

The intrinsic versus extrinsic proof style has implications for development, but both styles have advantages. An intrinsic proof style makes it easier to update code and proofs together, while the extrinsic style results in cleaner, "normal looking" source code. The extrinsic style also makes it easier to have multiple specifications for the same function. Likewise, the functionalization has its advantages and disadvantages. Pure functions are generally easy to reason about, allowing equational reasoning principles and so forth. However, it is not clear how, if at all, unsafe functions (like pointer accesses) or concurrent code can be functionalized.

Kani [76] is a Rust verifier based on bounded model checking (BMC). Because it is based on BMC, it cannot verify code that uses unbounded loops. Kani supports the checking of unsafe code, but it does not check for all possible occurrences of undefined behavior, and it does not handle concurrency.

Mendel [66] presents a verification approach for types with interior mutability that is more precise than what Verus can handle without ghost types. Mendel uses its notion of *implicit capabilities* to automatically deduce situations where an interiorly-mutable value cannot change. As a result, it can verify that consecutive reads of a cell return the same results, whereas Verus does not allow such a deduction for its invariant-based types like `InvCell` (§3.5.2) or `LocalInvariant`, not even when they are exclusively owned. To achieve the same result as Mendel, a Verus developer would need to track the data separately in some owned ghost type, a far more manual process than in Mendel.

RustBelt [33] is a foundational work in the Rust verification space. It claims not only the verification of Rust code, but the verification of Rust *type safety*. That is, RustBelt can show that *any* program, if it is well-typed in Rust's type system, will exhibit no undefined behavior. Furthermore, it can verify that functions which use unsafe code *internally* are correct, in the sense that any well-typed Rust program with access to these functions but which uses no *additional* unsafe code will have no undefined behavior. RustBelt uses the Iris separation logic, and it is mechanized in Coq. By contrast, Verus (and most other Rust verification tools) rely on Rust's type system but cannot reason directly about it; i.e., this sort of "for all programs" property is entirely out-of-scope.

RustBelt's model language λ_{Rust} is a simplification of real Rust in many ways, though it does capture most of what is important about ownership and lifetimes. Thus far, no version of RustBelt captures the entirety of Rust's memory ordering model—the original RustBelt handles *sequentially consistent* atomics, while follow-up work [14] handles relaxed memory. However, handling relaxed and sequentially-consistent *together* remains open. (In contrast, Verus only handles sequentially consistent ordering.) Specifying Rust in its entirety is a monumental task, and many elements of Rust's semantics, especially related to unsafe code, remain undefined.

RustBelt has also been extended to **RustHornBelt** [56] which verifies (a simplification of)

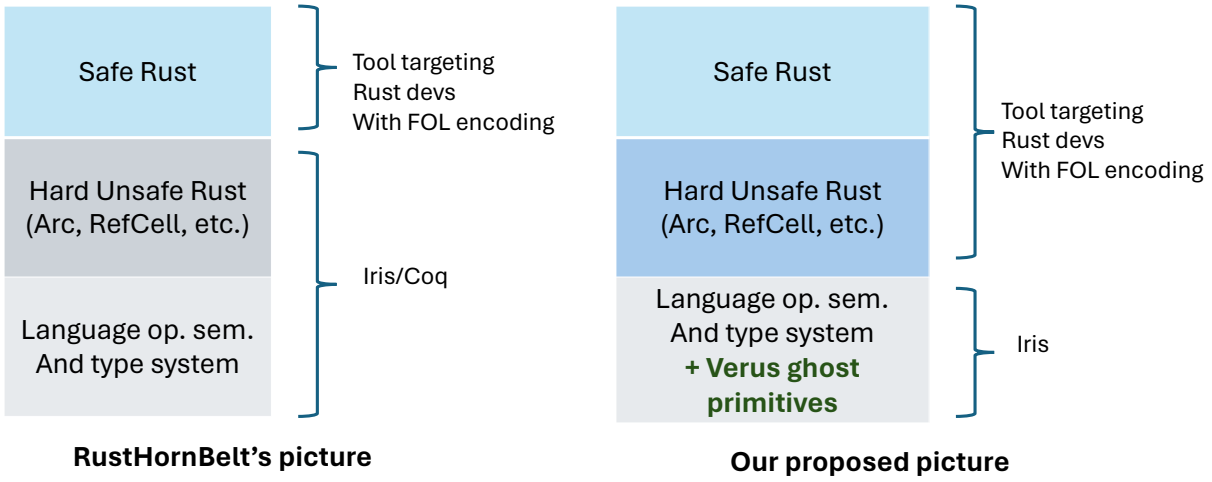


Figure 10.1: **RustHornBelt picture vs. Verus picture.**

the encoding used by Creusot. RustHornBelt proposes a *complementary* approach: RustHornBelt can be used to verify unsafe code (in Coq), while a first-order logic tool like Creusot can be used to verify safe "application code." **Gillian-Rust** [80] proposes a similar split, albeit using Gillian instead of Coq to verify unsafe code.

In this vein, this thesis proposes that we can push the "boundary" down (Figure 10.1), so that more unsafe code can be captured by the higher-level tool, in our case, Verus. Reasoning about the type system still remains in the domain of Iris, as we sketched in Chapter 6.

RefinedRust [21] is an approach for verifying Rust code foundationally. It fills in some of the gaps with λ_{Rust} to form a more realistic operational semantics which they call *Radium*. RefinedRust allows the user to annotate Rust programs with specifications that are translated into Coq. This contrasts with Verus and some of the other tools: Whereas Verus trusts the results of rustc's type-checker and applies metatheory that relies on Rust's type system, these things are all outside of the TCB for RefinedRust. Specifically, RefinedRust does *use* the results of rustc's type checker, but it does not *trust* that these results are correct, instead using the lifetime logic in Coq to check the results. There are trade-offs for this foundationalness: RefinedRust reports 22 minutes of CPU time to verify `Vec`—about 120 lines of code. Furthermore, RefinedRust does not currently support concurrency.

The **Stacked Borrows** [36] work and subsequent **Tree Borrows** [77] are efforts to define unsafe aliasing-related aspects of Rust's operational semantics more precisely. They have been used to verify some compiler optimizations, but to my knowledge, they have yet to be used as the basis for Rust program verification, which makes Radium the most precise model yet used for foundational verification of Rust programs.

Reliance on special types One limitation of Verus is that Verus is kind of a "dialect" of Rust where all of these applications need to use the special Verus primitive types. By contrast, most other Rust verification tools have greater focus on Rust "as is."

10.4 Systems verification

IronFleet and VeriBetrKV IronFleet [29] is a Dafny framework for verifying distributed systems. Even though this thesis does not cover distributed systems, IronFleet had much influence on IronSync's GSM system, as we have already discussed (Chapter 7). More specifically, IronFleet influenced the design of VeriBetrKV [25], which influenced IronSync.

Like in IronFleet, the key idea in IronSync is to first connect the program to some state machine abstraction and then prove things about the state machine. However, the way this connection works is very different.

In IronFleet, the program must be structured as loop that repeatedly calls into some "handler," and the handler has to prove that its pre- and post-states are related by some transition of the abstract state machine. Since this does not constrain its intermediate states at all, we must also argue that this handler can be treated *atomically*. The way IronFleet does this is by applying something called a *reduction argument* [52]. To do this, IronFleet needs to impose certain restrictions on the way the handler interacts with the environment. For example, in the networked setting, all nodes are communicating over the network, and the imposed restriction takes the form: "A single invocation of the handler can only perform message-receives followed by message-sends, not in the other order."

By contrast, IronSync removes the need for the reduction argument entirely. In IronSync, transitions are performed via actions on ghost state, and thus they occur "instantaneously," rather than being expressed as relations between pre- and post-states of a program execution. As a result, we can remove the "handler" system and simply have the implementation schedule its transitions internally. Finally, the tokenization system reduces the need to reason about the evolution of global state within the program.

Notably, the developers of IronFleet were able to prove liveness of some of their distributed systems. This was possible because liveness on the state machine abstractions implied liveness of the implementation. This is not true in IronSync. Proving termination in the multi-threaded setting is substantially harder than in the single-threaded setting, and it is not something I have yet attempted with this collection of techniques.

Perennial, GoJournal, and DaisyNFS The Perennial framework [8] is a recent research direction that culminated in the design of a verified file system, DaisyNFS [10]. DaisyNFS uses a hybrid approach: First, a crash-safe journaling system, GoJournal [9], is implemented in Go and proved correct using the Perennial framework (itself written in Iris). Second, DaisyNFS is implemented in Dafny, which compiles to Go and calls into GoJournal. The advantage of this approach is that the developer is able to use each tool to do what it does best. That is, they use Perennial/Iris to do the difficult parts related to storage persistence and concurrency, and they use Dafny for "high level" file system logic.

Verus is generally suitable to do most tasks Dafny can do, so the more interesting comparison is in the techniques used to build GoJournal and in particular, the elements related to the storage system and crash safety.

In Perennial, these aspects are all handled within the program logic. For example, Perennial introduces points-to propositions *for the disk*, i.e., propositions that say that a disk has a certain value at a certain address. Furtheremore, it introduces concepts of crash invariants and crash

specifications so that the program logic can reason about crash-recovery. We have seen much value in the points-to capabilities, so this is an obvious advantage of Perennial's approach. The GSM approach does not reason about the disk this way, and as a result, the disk-relevant parts of our refinement proof frequently reason like: "We are modifying disk page d, and all these other invariants only reference disk pages that are *not* d; therefore, all these other invariants still hold." This is exactly the sort of thing that is obviated by use of separation logic techniques.

On the other hand, specifying the behavior of the disk and the environment outside of the program logic makes it easier to modify the environment model without revamping the entire program logic. VeriBetrKV illustrated this when we modified the disk model to handle limited forms of corruptions. By contrast, if we reasoned about the disk with points-to assertions, making such changes could fundamentally change the program logic.

Chapter 11

Conclusion

We presented a methodology for verifying advanced concurrent systems.

Our key idea was to combine the power of efficient, automated type systems for managing data ownership with powerful ideas from the rich resource logics of modern CSL. We took the premise that these ideas would have excellent synergy and ran with it. We started by attempting to "copy-and-paste" algebraic laws from CSL as axioms in a highly-automated verification language based on ownership, but we quickly ran into interactions that needed new explanations. By developing out the theory to handle these interactions, we both shone a new light on some important ideas in semantic type soundness proofs and introduced powerful techniques into our verification language. As a result, we were able to tackle a number of sophisticated case studies.

The effectiveness of our approach to verification lies not just in theory, but also from an intense focus on scalability and practicality. Verus is useful not *only* because it has neat algebraic laws and a novel ghost state description language, but also from all the work my teammates have put into making it fast and capable. Rust's effort to bring ownership types into a mainstream language deserves a share of credit as well.

Of course, the journey is far from over, as their are plenty of questions to address. Can these techniques scale up to an entire key-value store? What about an operating system? A standard library?

Can we use the techniques to verify existing code, *without* having to "rewrite it in Verus"? How can we improve diagnostics for SMT-based verification? Can we make the approach more foundational without compromising on its efficiency? Can we bring some of our insights back into the CSL world?

I'm excited to see where these questions will take us. Formal verification is more exciting than ever, with mountains of foundational theory and fresh ideas for efficient, automated verification. Taking advantage of both, we can reach new heights.

Bibliography

[1] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016. doi: 10.1145/2872362.2872404. 10.1

[2] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA): 147:1–147:30, 2019. doi: 10.1145/3360573. URL https://doi.org/10.1145/3360573. 1.1, 10.3

[3] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. NrOS: Effective replication and sharing in an operating system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2021. ISBN 978-1-939133-22-9. 2.4

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. https://hal.inria.fr/hal-00790310. 10.3

[5] Richard Bornat, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. volume 40, pages 259–270, 01 2005. doi: 10.1145/1047659.1040327. 3.6.3, 4.5

[6] John Boyland. Checking interference with fractional permissions. pages 55–72, 06 2003. ISBN 978-3-540-40325-8. doi: 10.1007/3-540-44898-5_4. 4.5

[7] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037721. URL http://doi.acm.org/10.1145/3037697.3037721. 2.4, **??**

[8] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2019. 10.2.7, 10.4

[9] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai

Zeldovich. GoJournal: A verified, concurrent, crash-safe journaling system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/chajed. 9.5.3, 10.4

[10] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2022. 10.4

[11] Alex Conway, Abhishek K. Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard P. Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020. 1.3, 2.3, **??**

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing*, 2010. 9.5.1

[13] F.J. Corbató and Project MAC (Massachusetts Institute of Technology). *A Paging Experiment With The Multics System*. Project MAC. Massachusetts Institute of Technology, 1968. URL https://books.google.com/books?id=5wDQNwAACAAJ. 2.3

[14] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. Rustbelt meets relaxed memory. *Proc. ACM Program. Lang.*, 4(POPL):34:1–34:29, 2020. doi: 10.1145/3371102. URL https://doi.org/10.1145/3371102. 10.3

[15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover. In *Proceedings of the Conference on Automated Deduction (CADE)*, August 2015. 2.5, 10.3

[16] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24. 3.3.3

[17] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. SINGULAR 4-3-0 — A computer algebra system for polynomial computations. http://www.singular.uni-kl.de, 2022. 3.3.3

[18] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for the deductive verication of Rust programs. In *Proceedings of ICFEM 2022 - International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science, Madrid, Spain, October 2022. Springer Verlag. doi: 10.1007/978-3-031-17244-1_6. URL https://hal.inria.fr/hal-03737878. 1.1, 3.4.1, 10.3

[19] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. doi: 10.1145/360933.360975. URL https://doi.org/10.1145/360933.360975. 3.3.1

[20] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. Steel: proof-oriented programming in a dependently typed concurrent separation logic. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi: 10.1145/3473590. URL https://doi.org/10.1145/3473590. 10.2.3

[21] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. Refinedrust: A type system for high-assurance verification of Rust programs. *Proc. ACM Program. Lang.*, 8(PLDI), jun 2024. doi: 10.1145/3656422. URL https://doi.org/10.1145/3656422. 9.5.3, 10.3

[22] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi: 10.1016/0304-3975(87)90045-4. URL https://doi.org/10.1016/0304-3975(87)90045-4. 10.1

[23] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 2, pages 52–66, Pisa, Italy, March 1987. Springer-Verlag LNCS 250. 10.1

[24] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: an extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 653–669, USA, 2016. USENIX Association. ISBN 9781931971331. 9.5.3

[25] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 99–115. USENIX Association, November 2020. URL https://www.usenix.org/conference/osdi20/presentation/hance. 1.3, 1.4, 7.1, 7.2, 8.2, 9.5.3, 9.5.3, 10.4

[26] Travis Hance, Jon Howell, Oded Padon, and Bryan Parno. Leaf: Modularity for temporary sharing in separation logic. *Proc. ACM Program. Lang.*, 7(OOPSLA2), 2023. doi: 10.1145/3622798. URL https://doi.org/10.1145/3622798. 1.3, 1.4, 1, 4.5, 4.6

[27] Travis Hance, Jon Howell, Oded Padon, and Bryan Parno. Leaf: Modularity for temporary sharing in separation logic (extended version). 2023. doi: 10.48550/arXiv.2309.04851. URL https://arxiv.org/abs/2309.04851. 6.4.10

[28] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. Sharding the state machine: Automated modular reasoning for complex concurrent systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 911–929. USENIX Association, July 2023. URL https://www.usenix.org/conference/osdi23/presentation/hance. 1.3, 1.4, 3.1, 7.1, 9.23, 9.5.1, 9.24

[29] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17. ACM, 2015. doi: 10.1145/2815400.2815428. URL https://doi.org/10.1145/2815400.2815428. 1.3, 7.2, 9.5.3, 9.5.3, 9.5.3, 10.4

[30] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, 6(ICFP):711–741, 2022. doi: 10.1145/3547647. URL https://doi.org/10.1145/3547647. 1.1, 10.3

[31] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 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, page 637–650, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2676980. URL https://doi.org/10.1145/2676726.2676980. 1.3, 10.2.4

[32] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 256–269, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342193. doi: 10.1145/2951913.2951943. URL https://doi.org/10.1145/2951913.2951943. 1.3

[33] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL): 66:1–66:34, 2018. doi: 10.1145/3158154. URL https://doi.org/10.1145/3158154. 6, 6.2, 6.11, 10.2.5, 10.3

[34] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the Rust programming language — technical appendix and coq development, 2018. URL https://plv.mpi-sws.org/rustbelt/popl18/. 6.3.1

[35] Ralf Jung, R. Krebbers, Jacques-Henri Jourdan, A. Bizjak, L. Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018. 1.1, 3.2, 4.2, 4.1

[36] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for Rust. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371109. URL https://doi.org/10.1145/3371109. 10.3

[37] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371113. URL https://doi.org/10.1145/3371113. 9.6, 1, 10.2.6

[38] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010. doi: 10.1145/1743546.1743574. URL https://doi.org/10.1145/1743546.1743574. 9.5.3

[39] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. *SIGPLAN Not.*, 52(1):205–217, jan 2017. ISSN 0362-1340. doi: 10.1145/3093333.3009855. URL https://doi.org/10.1145/3093333.3009855. 10.2.3

[40] Shuvendu K. Lahiri, Shaz Qadeer, and David Walker. Linear maps. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*, 2011. 10.1.1

[41] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 01 1964. ISSN 0010-4620. doi: 10.1093/comjnl/6.4.308. URL https://doi.org/10.1093/comjnl/6.4.308. 6.5.1

[42] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), 2023. doi: 10.1145/3586037. URL https://doi.org/10.1145/3586037. 1.3, 1.4, 3.3.2, 3.7, 6.1

[43] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types (extended version). 2023. doi: 10.48550/ARXIV.2303.05491. URL https://arxiv.org/abs/2303.05491. 6.5.2

[44] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jay Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. OSDI (to appear), 2024. 1.3, 1.4, 9.25, 9.5.3

[45] Daan Leijen. Koka: Programming with row-polymorphic effect types. Technical Report MSR-TR-2013-79, Microsoft, August 2013. URL https://www.microsoft.com/en-us/research/publication/koka-programming-with-row-polymorphic-effect-types/. 2.5

[46] Daan Leijen. Mimalloc-bench. https://github.com/daanx/mimalloc-bench, 2023. 9.5.1

[47] Daan Leijen, Ben Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. Technical Report MSR-TR-2019-18, Microsoft, June 2019. URL https://www.microsoft.com/en-us/research/publication/mimalloc-free-list-sharding-in-action/. 2.5, ??, 9.3.3

[48] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20. URL https://doi.org/10.1007/978-3-642-17511-4_20. 3.1, 3.3.1

[49] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 378–393, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-00590-9. 10.2.2

[50] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. *Proc. ACM Program. Lang.*, 6 (OOPSLA1), 2022. doi: 10.1145/3527313. URL https://doi.org/10.1145/3527313. 10.1

[51] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. *Proc. ACM Program. Lang.*, 6 (OOPSLA):1–28, 2022. doi: 10.1145/3527313. URL https://doi.org/10.1145/3527313. 1.3, 6

[52] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun.*

ACM, 18(12):717–721, dec 1975. ISSN 0001-0782. doi: 10.1145/361227.361234. URL https://doi.org/10.1145/361227.361234. 10.4

[53] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Haojun Ma, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Automated verification of concurrent code with sound semantic extensibility. *ACM Transactions on Programming Languages and Systems*, 44(2), June 2022. 9.5.3

[54] Nicholas D. Matsakis and Felix S. Klock. The Rust language. *Ada Lett.*, 34(3):103–104, October 2014. ISSN 1094-3641. doi: 10.1145/2692956.2663188. URL https://doi.org/10.1145/2692956.2663188. 1.1

[55] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. RustHorn: CHC-based verification for Rust programs. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *LNCS*, pages 484–514. Springer, 2020. doi: 10.1007/978-3-030-44914-8_18. URL https://doi.org/10.1007/978-3-030-44914-8_18. 3.4.1, 10.3

[56] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 841–856. ACM, 2022. doi: 10.1145/3519939.3523704. URL https://doi.org/10.1145/3519939.3523704. 6.2, 10.3

[57] Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in Iris. In Luís Caires, editor, *Programming Languages and Systems*, pages 3–29, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17184-1. 6.4.9

[58] Greg Morrisett, Amal Ahmed, and Matthew Fluet. L3: A linear language with locations. In *Typed Lambda Calculi and Applications*, 2005. doi: 10.1007/11417170_22. 10.1.1

[59] Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: Automated verification of fine-grained concurrent programs in Iris. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2022. ISBN 9781450392655. doi: 10.1145/3519939.3523432. URL https://doi.org/10.1145/3519939.3523432. 10.2.3

[60] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *LNCS*, pages 41–62. Springer, 2016. doi: 10.1007/978-3-662-49122-5_2. URL https://doi.org/10.1007/978-3-662-49122-5_2. 10.2.2, 10.2.3, 10.2.5, 10.3

[61] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375 (1–3):271–307, April 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.12.035. URL https://doi.org/10.1016/j.tcs.2006.12.035. 1.1, 3.2

[62] Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, may 1976. ISSN 0001-0782. doi: 10.1145/360051.360224. URL https://doi.org/10.1145/360051.360224. 3.6.1

[63] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999. doi: 10.2307/421090. 4.2

[64] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 614–630. Association for Computing Machinery, 2016. doi: 10.1145/2908080.2908118. URL https://doi.org/10.1145/2908080.2908118. 5.3

[65] Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 439–458, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19718-5. 10.2.2

[66] Federico Poli, Xavier Denis, Peter Müller, and Alexander J. Summers. Reasoning about interior mutability in Rust using library-defined capabilities, 2024. 10.3

[67] Upamanyu Sharma. verus-grove. https://github.com/upamanyus/verus-grove/blob/main/false.rs. Accessed: 2024-08-09. 6.5.1

[68] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 148–172, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03013-0. 10.2.2

[69] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022. doi: 10.1145/3547631. URL https://doi.org/10.1145/3547631. 6.5.2

[70] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2016. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837655. URL http://dx.doi.org/10.1145/2837614.2837655. 3.3.1

[71] The Iris Team. The Iris 4.2 reference. https://plv.mpi-sws.org/iris/appendix-4.2.pdf. Accessed: 2024-08-09. 6.5.1

[72] The Iris Team. Iris 4.2. https://gitlab.mpi-sws.org/iris/iris/-/blob/dc6b798d8b8ab8937e8583b813f75560c2d833d5/iris/algebra/sts.v, 2023. 10.2.4

[73] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. Trillium: Higher-order concurrent and distributed separation logic for intensional refinement. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024. doi: 10.1145/3632851. URL https://doi.org/10.1145/3632851. 10.2.7

[74] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. A logical approach to type soundness. *Journal of the ACM (JACM), to appear*, 2024. 6.4

[75] Aaron Turon, Derek Dreyer, and Lars Birkedal. 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, page 377–390, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323260. doi: 10.1145/2500365.2500600. URL https://doi.org/10.1145/2500365.2500600. 10.2.4, 10.2.7

[76] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, page 321–330, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392266. doi: 10.1145/3510457.3513031. URL https://doi.org/10.1145/3510457.3513031. 10.3

[77] Neven Villani. Tree borrows. https://perso.crans.org/vanille/treebor/. Accessed: 2024-06-29. 10.3

[78] Philip Wadler. Linear types can change the world! In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods*, 1990. 10.1

[79] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. GhostCell: Separating permissions from data in Rust. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi: 10.1145/3473597. URL https://doi.org/10.1145/3473597. 10.1.1

[80] Sacha Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. A hybrid approach to semi-automated Rust verification, 2024. 10.3