Leaf: Modularity for Temporary Sharing in Separation Logic

TRAVIS HANCE, Carnegie Mellon University, USA
JON HOWELL, VMware Research, USA
ODED PADON, VMware Research, USA
BRYAN PARNO, Carnegie Mellon University, USA

In concurrent verification, separation logic provides a strong story for handling both resources that are owned exclusively and resources that are shared persistently (i.e., forever). However, the situation is more complicated for temporarily shared state, where state might be shared and then later reclaimed as exclusive. We believe that a framework for temporarily-shared state should meet two key goals not adequately met by existing techniques. One, it should allow and encourage users to verify new sharing strategies. Two, it should provide an abstraction where users manipulate shared state in a way agnostic to the means with which it is shared.

We present Leaf, a library in the Iris separation logic which accomplishes both of these goals by introducing a novel operator, which we call *guarding*, that allows one proposition to represent a shared version of another. We demonstrate that Leaf meets these two goals through a modular case study: we verify a reader-writer lock that supports shared state, and a hash table built on top of it that uses shared state.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: concurrent separation logic, verification, fractional permissions, counting permissions, reader-writer lock, read-sharing

ACM Reference Format:
Travis Hance, Jon Howell, Oded Padon, and Bryan Parno. 2023. Leaf: Modularity for Temporary Sharing in Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 223 (October 2023), 28 pages. https://doi.org/10.1145/3622798

1 INTRODUCTION

Multi-threaded concurrent programs are difficult to get right. One challenging pattern in such programs is *read-sharing*, i.e., allowing multiple threads to simultaneously read mutable shared state as long as no other thread is actively writing. This common optimization reduces thread contention and is often considered critical for scaling concurrent performance. While the general idea is commonly deployed, the concrete instantiations vary wildly. For example, even the implementation of a conceptually simple reader-writer lock grows quite complicated as various kinds of scaling issues are considered [Calciu et al. 2013; Dice and Kogan 2019; Guerraoui et al. 2019; Hsieh and Weihl 1992; Kashyap et al. 2017; Liu et al. 2014; Shirako et al. 2012]. As a concrete instance, one possible optimization uses multiple reference counters, each on its own cache line, to reduce thread contention for readers [Calciu et al. 2013]. And yet the challenge does not stop at reader-writer locks; for instance, the node-replication algorithm of Calciu et al. [2017] might allow simultaneous read-access to particular entries of a ring buffer, and this protocol does not resemble a lock in the

Authors' addresses: Travis Hance, thance@andrew.cmu.edu, Carnegie Mellon University, Pittsburgh, USA; Jon Howell, howell@vmware.com, VMware Research, Bellevue, USA; Oded Padon, oded.padon@gmail.com, VMware Research, Palo Alto, USA; Bryan Parno, parno@cmu.edu, Carnegie Mellon University, Pittsburgh, USA.

2475-1421/2023/10-ART223
https://doi.org/10.1145/3622798

slightest. Given all of this complexity, we would naturally like to verify, in a modular fashion, both the read-sharing implementations and the programs that use them.

One effective tool for reasoning about concurrent programs is *concurrent separation logic (CSL)* [O'Hearn 2007; Reynolds 2002] which lets us reason naturally about exclusive ownership of memory and, more generally, arbitrary resources. This is a great fit for (mutually exclusive) locks: it is easy to write a specification of a lock that allows the client to obtain ownership of a resource (e.g., permission to access some part of memory, or a more complicated invariant) which is returned upon releasing the lock, allowing the resource to be transferred between threads.

But how do we handle simultaneous read-sharing? We want some way of reasoning about this *simultaneous, shared* access—e.g., we want to talk about memory access where we can only read but not write, or invariants which must be preserved until the shared access is released. CSL's exclusive ownership does not work here, since the desired type of ownership is not exclusive. Some more recent CSLs also have a concept called *persistent knowledge* [Bizjak and Birkedal 2018], which allows indefinite sharing, but this on its own does not suffice either: whatever sharing mechanism we use needs a way to *reclaim* exclusive access once all shared access has been revoked.

Two of the earliest proposed methods to represent shared state while allowing reclamation are *fractional permissions* [Boyland 2003] and *counting permissions* [Bornat et al. 2005]. These representations have the benefit of being easy to construct and prove sound, but they do not form complete proof strategies for complex sharing protocols like the above, which are likely to have considerably more state that evolves in complex ways and cannot be expressed through fractional permissions or reference counters alone. Furthermore, the desire for modularity means there is pressure for specifications to converge on one particular representation so that they can interoperate; this limits flexibility since different representations might be more suitable for different applications. How can we support a wide gamut of sharing techniques—so the user can choose the right tool for the job, even building their own representation if necessary—while also achieving modularity?

Our approach is to take the core idea of fractional and counting permissions, generalize it, and a provide a uniform way to reason about it. That core idea, we argue, is that both involve propositions which are able to "stand in" for some kind of shared resource, but which are suitable for manipulation within the substructural separation logic. This enables them to handle the temporality of the sharing. This motivates us to extract the essence of this "standing-in" relationship and brings us to our contribution, the Leaf logic.

Leaf introduces a novel operator ⤖, which we call *guarding*, to represent the relationship between an exclusively owned proposition and the shared proposition it represents. Leaf handles both sides of this abstraction. First, we show how the user can *deduce* nontrivial ⤖ relationships by constructing arbitrary sharing protocols. Such protocols include ones based on the above-mentioned patterns, as well as custom protocols that are tailored to particular implementations. Our approach is inspired by *custom ghost state* [Dinsdale-Young et al. 2013, 2010; Jung et al. 2015; Krishnaswami et al. 2012; Ley-Wild and Nanevski 2013; Nanevski et al. 2014], a class of flexible separation logic techniques. We call the protocols of our new formulation *storage protocols*. Second, we show how to *make use of* ⤖ relationships, through general rules that are agnostic to the underlying sharing mechanism, thus enabling modular specifications. This approach is symbiotic between ghost state and read-sharing: by applying custom-ghost-state techniques, Leaf supplies a general form for read-sharing mechanisms; meanwhile, some ghost-state constructions become simpler because they can rely on Leaf's read-sharing, without needing to include their own bespoke sharing mechanisms.

Storage protocols can support complex algorithms found in real-world concurrent software systems. As we discuss in §6, Leaf's storage protocols have already been used in the IronSync framework [Hance et al. 2023c], which is enabled by Leaf's systematic approach to read-shared custom ghost state. IronSync targets production-scale, high-performance concurrent systems, e.g., a multi-threaded page cache (reaching 3M ops / second) or the node-replication algorithm mentioned above (3M ops / second across 192 threads). These results confirm that high-performance applications contain sophisticated, domain-specific read-sharing patterns; demonstrate that Leaf's storage protocols can handle them; and provide evidence that system developers find Leaf's perspective on temporarily read-shared resources useful. In this paper, though, we will mostly focus on the technical formalism of this perspective, so we use a smaller, self-contained example that is simple enough to explain in full, but still complex enough to show the utility of Leaf.

Specifically, we make the following contributions:

- We present Leaf, which has built-in deduction rules for temporarily shared state and a mechanism for user-defined sharing protocols based on ghost state, which we call *storage protocols*.
- We show how storage protocols capture existing patterns, e.g., fractional and counting permissions.
- We illustrate Leaf's modular specifications through a case study of a reader-writer lock and a hash table, demonstrating several different facets of sharing:
  - The hash table itself is shared between threads.
  - The hash table is composed of many reader-writer locks, which inherit that sharedness.
  - The memory cells in the reader-writer lock further inherit that sharedness, and they are accessed atomically by multiple threads.
  - The reader-writer locks allow temporary, shared read-only access to the hash table's memory slots. Furthermore, this is done via a clean, modular specification of the reader-writer lock.
- We prove the soundness of Leaf and provide it as a library in the Iris framework [Jung et al. 2018] in Coq. We mechanize our case studies in Coq. We mechanize our case studies in Coq. These proofs are available in our artifact [Hance et al. 2023a].

2 OVERVIEW

Leaf is constructed as a library in the *Iris separation logic* [Jung et al. 2018]. We generally use Iris notation, where applicable, and all the standard Iris proof rules apply. We assume familiarity with separation logic basics (the connectives $*$, $-\!\!*$, and so on), but we will review key Iris features as they come up.

2.1 Leaf Introduction: Resource Guarding

The primary question we need to unravel is how to talk generally about "a shared P" for any proposition P. Here, the proposition P might be something simple, like the permission to access a certain memory location ℓ and read a specific value v, denoted $\ell \hookrightarrow v$, or it might be a more complex invariant. In order to make shared state reclaimable, we connect the "shared P" to some exclusively owned (not persistent) proposition.

We do this via a relationship $G \rightarrowtail\!\!\!\!\rightarrow P$, pronounced G *guards* P. $G \rightarrowtail\!\!\!\!\rightarrow P$ 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 $G \rightarrowtail\!\!\!\!\rightarrow P$ 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.

In general, then, when we want to write a proof that operates over some read-only P in a way that abstracts over the way P is shared, we can write the proof to take ownership of some arbitrary Iris proposition $G : \mathit{iProp}$ where $G \rightarrowtail\!\!\!\!\rightarrow P$. To codify this pattern, we use a shorthand, $[X]\ \{P\}\ e\ \{Q\}$,

RwLock Specification

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

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

$$\begin{array}{rrll}
\forall F, x. & \{F(x)\} & \mathsf{rwlock\_new}() & \{rw.\ \exists \gamma, \mathsf{IsRwLock}(rw, \gamma, F)\} \\
\forall rw, \gamma, F. & \{\mathsf{IsRwLock}(rw, \gamma, F)\} & \mathsf{rwlock\_free}(rw) & \{\} \\
\forall rw, \gamma, F.\ [\mathsf{IsRwLock}(rw, \gamma, F)] & \{\} & \mathsf{lock\_exc}(rw) & \{\mathsf{Exc}(\gamma) * \exists x.\ F(x)\} \\
\forall rw, \gamma, F, x.\ [\mathsf{IsRwLock}(rw, \gamma, F)] & \{\mathsf{Exc}(\gamma) * F(x)\} & \mathsf{unlock\_exc}(rw) & \{\} \\
\forall rw, \gamma, F.\ [\mathsf{IsRwLock}(rw, \gamma, F)] & \{\} & \mathsf{lock\_shared}(rw) & \{\exists x.\ \mathsf{Sh}(\gamma, x)\} \\
\forall rw, \gamma, F, x.\ [\mathsf{IsRwLock}(rw, \gamma, F)] & \{\mathsf{Sh}(\gamma, x)\} & \mathsf{unlock\_shared}(rw) & \{\}
\end{array}$$

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

Fig. 1. Example specification for a reader-writer lock using Leaf notation. In §4, we show how to prove this specification for a particular implementation.

to mean $\forall G : \mathit{iProp}.\ \{P * G * (G \mathrel{\rightarrowtail\!\!\!\!\twoheadrightarrow} X)\}\ e\ \{Q * G\}$. 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." We indicate shared resources in purple, though this is only a visual aid, and it has no syntactic meaning.

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
$$\{\ell \hookrightarrow v\}\ \ell \leftarrow v'\ \{\ell \hookrightarrow v'\}$$

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

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.

2.2 Example: A Reader-Writer Lock Specification

The reader-writer lock spec in Figure 1 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* $F : X \longrightarrow \mathit{iProp}$ 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 $\rightarrowtail\!\!\!\rightarrow$ 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 $\mathsf{Sh}(\gamma, x) \rightarrowtail\!\!\!\rightarrow F(x)$. 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. We view Sh as a separation logic analogue of a *lock guard*, an object in some locking APIs [The cppreference Team 2011; The Rust Team 2014] which exists for the duration of a held lock. Indeed, this inspires the *guarding* name.

Notice the choice of parameter set X and proposition family F determines exactly what it means to be "read-only," because it is the value $x : X$ which is fixed until the client releases the shared lock. For example, we might set $X = \mathbb{Z}$ and $F = \lambda x : \mathbb{Z}.\ \ell \hookrightarrow x$. Then the client can take a shared lock and obtain shared $\ell \hookrightarrow x$ for some fixed integer x, which cannot change until the lock is released. On the other hand, they might set, say, $X = \mathbb{Z}_2$ and $F = \lambda x : \mathbb{Z}_2.\ \exists n.\ (\ell \hookrightarrow n) * (n = x \mod 2)$. In this case, upon taking the shared lock, they receive $\ell \hookrightarrow v$, but now only the *parity* of v is fixed. In fact, in this situation, the user would be able to update v to another value of the same parity (provided they do so in an atomic operation).

The RwLock spec raises two questions: How can the client do interesting things when they have $\mathsf{Sh}(\gamma, x)$, i.e., a "shared $F(x)$", rather than exclusive ownership of $F(x)$? Secondly, how can we verify a realistic lock implementation against this spec, which requires the deduction of a nontrivial guard relationship $\mathsf{Sh}(\gamma, x) \rightarrowtail\!\!\!\rightarrow F(x)$? Let us tackle these in turn.

2.3 Utilizing Shared State

How does the user actually benefit from shared state, i.e., state (like $F(x)$) under a guard operator, as in $(\mathsf{Sh}(\gamma, x) \rightarrowtail\!\!\!\rightarrow F(x))$ from the previous example?

In general, if $G \rightarrowtail\!\!\!\rightarrow P$, Leaf aims to let G be usable in any operation that could have used P, provided that P is not modified. Such an operation might be given by the Iris operator called the *view shift*, as in $P * A \Rrightarrow P * B$. In general, the view shift ($\Rrightarrow$) effectively says we can give up the resources on the left side to obtain the resources on the right. In the example, though, with P on both the left and right sides, P is *not* consumed, although it *is* needed to perform the operation. In this case, we could use G in place of P; i.e., we would have $G * A \Rrightarrow G * B$.

Frequently, in order to perform such updates, we need to first compose multiple pieces of shared state together; for example, suppose we employ a fine-grained locking scheme, where a thread might hold multiple pieces of state from different locks in shared mode, which *all together* are needed to perform a certain update or deduction. Ordinarily, we would compose the corresponding propositions with separating conjunction ($*$), but here, the pieces, being shared, might come from the same source and not actually be separated. To get around this, we use overlapping conjunction ($\wedge$) rather than $*$ when dealing with shared state. It turns out that constructing sound deduction rules to use $\wedge$ is subtle; the rule we give in §3.2 requires a specific technical condition. We will show how all this works together through our fine-grained hash table example (§5).

2.4 Deducing Guard Relationships

Towards verifying an implementation of a reader-writer lock, the most salient technical question is how we can construct nontrivial propositions like $\mathsf{Sh}(\gamma, x)$ and prove guard relationships on them. To tackle this question, it helps to first look at simpler examples of nontrivial guard relationships. As such, let us take a look at *fractional permissions* [Boyland 2003] and *counting permissions* [Bornat et al. 2005], two of the oldest known methods used to account for reclaimable read-shared permissions for memory (and other resources).

Fractional Permissions

Propositions: $\ell \xhookrightarrow{\text{frac}}_q v$

(where $\ell : Loc,\ v : Value,\ q : \mathbb{Q}$, and $q > 0$)

$$(\ell \hookrightarrow v) \Lleftarrow\!\Rrightarrow_{\mathcal{F}rac} (\ell \xhookrightarrow{\text{frac}}_1 v)$$

$$(\ell \xhookrightarrow{\text{frac}}_q v) \twoheadrightarrowtail_{\mathcal{F}rac} (\ell \hookrightarrow v)$$

$$(\ell \xhookrightarrow{\text{frac}}_{q_1+q_2} v) \dashv\vdash (\ell \xhookrightarrow{\text{frac}}_{q_1} v) * (\ell \xhookrightarrow{\text{frac}}_{q_2} v)$$

Counting Permissions

Propositions: $\ell \xhookrightarrow{\text{count}}_n v \quad \ell \overset{\text{ro}}{\hookrightarrow} v$

(where $\ell : Loc,\ v : Value,\ n : \mathbb{N}$)

$$(\ell \hookrightarrow v) \Lleftarrow\!\Rrightarrow_{Count} (\ell \xhookrightarrow{\text{count}}_0 v)$$

$$(\ell \overset{\text{ro}}{\hookrightarrow} v) \twoheadrightarrowtail_{Count} (\ell \hookrightarrow v)$$

$$(\ell \xhookrightarrow{\text{count}}_n v) \dashv\vdash (\ell \xhookrightarrow{\text{count}}_{n+1} v) * (\ell \overset{\text{ro}}{\hookrightarrow} v)$$

Fig. 2. Fractional and counting permissions expressed by the $\twoheadrightarrowtail$ operator. The $\Lleftarrow\!\Rrightarrow$ means we can perform an *update* to exchange exclusive ownership of one side for the other, while $\dashv\vdash$ means both sides are equivalent. $\mathcal{F}rac$ and *Count* are arbitrary *namespaces* (§3.4). In Leaf, these laws are derived from storage protocols (§3.4).

2.4.1 Fractional and Counting Examples. In the fractional paradigm, the points-to proposition is labeled with a rational number $q : Q$. These propositions combine additively: $(\ell \xhookrightarrow{\text{frac}}_q v) * (\ell \xhookrightarrow{\text{frac}}_{q'} v) \dashv\vdash (\ell \xhookrightarrow{\text{frac}}_{q+q'} v)$, where the $\dashv\vdash$ is bidirectional entailment, i.e., the two sides are equivalent. Write permission is given by $\ell \xhookrightarrow{\text{frac}}_1 v$ and read permission is given by any $\ell \xhookrightarrow{\text{frac}}_q v$ where $q > 0$. The idea is that the $\ell \xhookrightarrow{\text{frac}}_1 v$ can be split into multiple fractional pieces, which can be handed out and used in a read-only fashion, and then put back together to obtain write access, allowing the user to change v. Intuitively, the reason this works is that one cannot reclaim write access without gathering *all* the read-only pieces, since all of them are needed to sum back to 1. Thus anyone holding onto a read-only piece cannot have the value changed out from under them by another thread.

Counting permissions, on the other hand, does not allow arbitrary splitting, but instead uses a centralized counter, $\ell \xhookrightarrow{\text{count}}_n v$ ($n : \mathbb{N}$) to keep track of the number of extant read-only permissions, denoted $\ell \overset{\text{ro}}{\hookrightarrow} v$. The user can increment the counter to obtain another read-only permission, or perform the inverse: $(\ell \xhookrightarrow{\text{count}}_n v) \dashv\vdash (\ell \xhookrightarrow{\text{count}}_{n+1} v) * (\ell \overset{\text{ro}}{\hookrightarrow} v)$. Meanwhile, $\ell \xhookrightarrow{\text{count}}_0 v$ gives write permission; i.e., we can write as long as there are zero read permissions in existence.

In Leaf, we can express both these patterns using $\twoheadrightarrowtail$, as shown in Figure 2. The idea of the "write permission" is expressed by saying that $\ell \xhookrightarrow{\text{frac}}_1 v$ can be exchanged for $\ell \hookrightarrow v$, and vice versa; the "read permission" is expressed by the guards relationship: $(\ell \xhookrightarrow{\text{frac}}_q v) \twoheadrightarrowtail_{\mathcal{F}rac} (\ell \hookrightarrow v)$. (We explain the $\mathcal{F}rac$ label later.) The same approach works for the counting permissions.

Note that with this setup, we do not need to prove the heap read and write rules for the fractional and counting permissions individually. Rather, we simply apply the more general Heap-Write and Heap-Read-Shared rules from earlier along with any guard relationship, such as one from Figure 2.

2.4.2 Nontrivial Guarding with Storage Protocols. Now, we return to our question from earlier: how can we, in general, soundly construct propositions like $(\ell \xhookrightarrow{\text{frac}}_q v)$, $(\ell \overset{\text{ro}}{\hookrightarrow} v)$, or $\mathsf{Sh}(\gamma, x)$? What are the primitive deduction rules for $\twoheadrightarrowtail$, and in particular, what are the rules that allow us to prove nontrivial $\twoheadrightarrowtail$ relations on those propositions?

All of these can be constructed by via a Leaf formalism called a *storage protocol*, so named because they allow the user to "store" propositions (e.g., by $(\ell \hookrightarrow v) \Rrightarrow_{\mathcal{F}rac} (\ell \xhookrightarrow{\text{frac}}_1 v)$) and then access them in a shared manner. The core idea is based off of *custom ghost state*, a concept wherein a user may define their own resources and derive update relations ($\Rrightarrow$). Storage protocols extend the concept to also allow the derivation of guard relations ($\twoheadrightarrowtail$).

The propositions constructed by the protocol are able to guard arbitrary propositions that have no intrinsic notion of being shareable. For example, in the fractional example, $\ell \hookrightarrow v$ has no notion of being shareable; rather, *given* $\ell \hookrightarrow v$, without knowing anything about its definition, Leaf allows us to construct the $\ell \xhookrightarrow{\text{frac}}_q v$ proposition with a particular guard relationship to $\ell \hookrightarrow v$. This is an instance of the same feature that lets us have $\mathsf{Sh}(\gamma, x) \twoheadrightarrowtail F(x)$ parameterized by an arbitrary proposition family F.

2.5 Outline

Throughout the paper, we explore these examples in more detail. We first formally introduce $\twoheadrightarrowtail$ and its elementary deduction rules, and sketch how we can derive rules like Heap-Read-Shared within Leaf. We then introduce our new formulation of custom ghost state, allowing the deduction of nontrivial $\twoheadrightarrowtail$ propositions, such as those in Figure 2. We show how to verify a reader-writer lock, proving the specification (Figure 1) holds for a particular implementation. To illustrate Leaf's modular specifications, we then build another application on top of the reader-writer lock. Finally, we discuss our construction of Leaf within Iris and the definition of $\twoheadrightarrowtail$, proving our laws sound.

3 THE LEAF LOGIC

We begin our presentation by reviewing the concept of custom ghost state. The formulation we present first is (largely) standard, yet still significant within Leaf. Then we will dive into Leaf's $\twoheadrightarrowtail$ operator and our new extension of custom ghost state.

3.1 Custom Ghost State (Background)

Custom ghost state in Iris is a mechanism through which the user can soundly construct their own resource with custom update rules. We present a simplified version of it here, based primarily on the "Iris 1.0" formulation [Jung et al. 2015].

The construction is parameterized by a *partial commutative monoid* (PCM) whose elements form the basis of the resource. Formally, a PCM is a set M (the *carrier* set) with a composition operator $\cdot : M \times M \longrightarrow M$ which is associative and commutative, and with a unit element ϵ. We let $a \preceq b \triangleq (\exists c.\ a \cdot c = b)$. The partiality is represented by a *validity predicate* $\mathcal{V} : M \longrightarrow \mathsf{Bool}$, where $\mathcal{V}(\epsilon)$ and $\forall a, b.\ a \preceq b \wedge \mathcal{V}(b) \Rightarrow \mathcal{V}(a)$. We also define a derived relation $\rightsquigarrow$ called the *frame-preserving update*,

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

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 shows the rules in Figure 3 are sound for a proposition written $\boxed{a}^{\gamma}$ for $a : M$. 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 PCM-Update. The operator $\Rrightarrow$ is called *view shift*, and it essentially means we can give up the resource on the left-hand to obtain the resource on the right. VS-Hoare says we can perform such updates at any program point during a proof. Note that the view shifts can also be annotated with a *mask*, denoted $\Rrightarrow_{\mathcal{E}}$; we discuss this further in the next section.

Example 3.1. An archetypal PCM 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:

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

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

Rules for PCM ghost state

Instantiated for a given PCM $(M, \cdot, \mathcal{V})$

Propositions: $\boxed{a}^{\gamma}$ (where $a : M,\ \gamma : \mathit{Name}$)

PCM-Unit
$$\text{True} \vdash \boxed{\epsilon}^{\gamma}$$

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

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

PCM-Alloc
$$\dfrac{\mathcal{V}(a)}{\text{True} \Rrightarrow \exists \gamma.\ \boxed{a}^{\gamma}}$$

PCM-Update
$$\dfrac{a \rightsquigarrow b}{\boxed{a}^{\gamma} \Rrightarrow \boxed{b}^{\gamma}}$$

PCM-And
$$\dfrac{\forall t.\ ((x \preceq t) \land (y \preceq t) \land \mathcal{V}(t)) \Rightarrow (z \preceq t)}{\boxed{x}^{\gamma} \land \boxed{y}^{\gamma} \vdash \boxed{z}^{\gamma}}$$

Basic Properties of View Shifts

VS-Hoare
$$\dfrac{P' \Rrightarrow_{\mathcal{E}} P \quad \{P\}\ e\ \{Q\}_{\mathcal{E}} \quad Q \Rrightarrow_{\mathcal{E}} Q'}{\{P'\}\ e\ \{Q'\}_{\mathcal{E}}}$$

VS-Weaken
$$\dfrac{P \Rrightarrow_{\mathcal{E}_1} Q}{P \Rrightarrow_{\mathcal{E}_1 \cup \mathcal{E}_2} Q}$$

Fig. 3. Deduction rules for PCM-based ghost state and view shifts.

Using the overlapping conjunction. We make a point to include a rule for overlapping conjunction, since in dealing with shared state we often have the potential for overlap. PCM-And lets us deduce that $\boxed{x}^{\gamma} \land \boxed{y}^{\gamma} \vdash \boxed{z}^{\gamma}$ under the premise that, if for all valid states t with x and y as sub-states, z is also a sub-state. As an example, consider $\boxed{\mathrm{ex}(x)}^{\gamma} \land \boxed{\mathrm{ex}(y)}^{\gamma}$. If $x \neq y$, then $\boxed{\mathrm{ex}(x)}^{\gamma} \land \boxed{\mathrm{ex}(y)}^{\gamma} \vdash \boxed{\lightning}^{\gamma}$ by PCM-And because *no* valid state includes both $\boxed{\mathrm{ex}(x)}^{\gamma}$ and $\boxed{\mathrm{ex}(y)}^{\gamma}$. Then we can conclude that $\boxed{\mathrm{ex}(x)}^{\gamma} \land \boxed{\mathrm{ex}(y)}^{\gamma} \vdash (x = y)$.

3.2 The Guarding Operator and Its Elementary Deduction Rules

The fundamental building block of the Leaf logic is the $⤖$ operator, pronounced *guards*. When we write $G ⤖_{\mathcal{E}} I$, we call G the *guard* and I the *guarded proposition* or sometimes the *guarded invariant*, and this means that having G allows shared access to I. Guards, like view shifts, are annotated with a mask $\mathcal{E}$, as we discuss below. The basic rules for $⤖_{\mathcal{E}}$ are given in Figure 4.

For example, Guard-Refl says that a P represents a shared P, while Guard-Trans says that if Q is a shared R, then a shared Q is also a shared R. Guard-Pers and Unguard-Pers show how persistent propositions can move into or out from under a $⤖$. Guard-Upd says that when an update $P * A \Rrightarrow P * B$ is valid, then we can perform this update even when we have a shared P.

Mask Sets. We use mask sets, $\mathcal{E}$, to track the "sources" of the sharing for both guard relations ($⤖_{\mathcal{E}}$) and view shifts ($\Rrightarrow_{\mathcal{E}}$). The sharing sources—as we will see later—are called storage protocols, and each storage protocol has a name γ. A mask is a set of such names, and it can be considered an over-approximation of the set of storage protocols involved in the derivation of a given relation. (In §7, we will see that this interpretation of $\Rrightarrow_{\mathcal{E}}$ follows from its usual Iris definition.)

We need to track these because we need to be careful when we apply Guard-Upd. If we applied Guard-Upd twice while ignoring its disjointness condition, we could potentially "double up" a proposition that is shared between multiple guard objects. For example, if we had $G_1 \Rrightarrow_{\mathcal{E}} P$ and $G_2 \Rrightarrow_{\mathcal{E}} P$, we could use $G_1 * G_2$ to perform an update that would only be possible if we had $P * P$. The disjointness condition prevents us from doing this. Those familiar with Iris may observe that this is similar to invariant reentrancy, so it should be no surprise that we solve the problem the same way, that is, via mask sets.

Guards and implications. One might expect a rule where we use $G ⤖ I$ and $I \vdash J$ to conclude $G ⤖ J$. This *does* work when we can write $I = J * J'$ for some J' (Guard-Split), but it does not hold in general: consider, for example, a judgment such as $(\ell \hookrightarrow 1) \vdash (\exists x.\ \ell \hookrightarrow x)$. It would be unsound

Deduction rules for guarded resources

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

Guard-Refl
$$P \twoheadrightarrowtail_{\mathcal{E}} P$$

Guard-Trans
$$(P \twoheadrightarrowtail_{\mathcal{E}} Q) * (Q \twoheadrightarrowtail_{\mathcal{E}} R) \vdash (P \twoheadrightarrowtail_{\mathcal{E}} R)$$

Guard-Split
$$P * Q \twoheadrightarrowtail_{\mathcal{E}} P$$

Guard-Weaken-Mask
$$(G \twoheadrightarrowtail_{\mathcal{E}_1} P) \vdash (G \twoheadrightarrowtail_{\mathcal{E}_1 \cup \mathcal{E}_2} P)$$

Guard-Pers
$$\frac{\mathsf{persistent}(C)}{(G \twoheadrightarrowtail_{\mathcal{E}} A) * C \vdash (G \twoheadrightarrowtail_{\mathcal{E}} A * C)}$$

Unguard-Pers
$$\frac{A * B \vdash C \quad \mathsf{persistent}(C)}{G * (G \twoheadrightarrowtail_{\mathcal{E}} A) * B \Rrightarrow_{\mathcal{E}} G * (G \twoheadrightarrowtail_{\mathcal{E}} A) * B * C}$$

Guard-Upd
$$\frac{\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset}{(G \twoheadrightarrowtail_{\mathcal{E}_1} P) * (P * A \Rrightarrow_{\mathcal{E}_2} P * B) \vdash (G * A \Rrightarrow_{\mathcal{E}_1 \cup \mathcal{E}_2} G * B)}$$

PointProp-Own
$$\mathsf{point}(\boxed{x}^{\gamma})$$

Guard-Implies
$$\frac{A \vdash P \quad \mathsf{point}(P)}{(G \twoheadrightarrowtail_{\mathcal{E}} A) \vdash (G \twoheadrightarrowtail_{\mathcal{E}} P)}$$

Guard-And
$$\frac{A \land B \vdash P \quad \mathsf{point}(P)}{(G \twoheadrightarrowtail_{\mathcal{E}} A) * (G \twoheadrightarrowtail_{\mathcal{E}} B) \vdash (G \twoheadrightarrowtail_{\mathcal{E}} P)}$$

PointProp-Sep
$$\frac{\mathsf{point}(P) \quad \mathsf{point}(Q)}{\mathsf{point}(P * Q)}$$

Fig. 4. Deduction rules for $\twoheadrightarrowtail$ introduced by Leaf.

if a user sharing $\ell \hookrightarrow 1$ could "downgrade" it to the right-hand side; that user could then update ℓ to a different value and invalidate the proposition the other users were relying on.

Interestingly, it turns out that there are some propositions J such that any judgment $I \vdash J$ *can* always be "split" into $I = J * J'$. Specifically, this happens whenever J is of the form $\boxed{x}^{\gamma}$ or a conjunction thereof. We call these *point propositions* and indicate them by $\mathsf{point}(J)$ (PointProp-Own, PointProp-Sep). For such J, we can indeed conclude $G \twoheadrightarrowtail J$ (Guard-Implies).

Overlapping Conjunction. How can we compose shared state? We certainly cannot have a rule like $(G \twoheadrightarrowtail_{\mathcal{E}} A) * (H \twoheadrightarrowtail_{\mathcal{E}} B) \vdash ((G * H) \twoheadrightarrowtail_{\mathcal{E}} (A * B))$. After all, A and B might be shared from the same source and thus not be properly separated. Somewhat surprisingly, this rule is not even sound if we require the masks to be disjoint. (See the extended version of our paper [Hance et al. 2023b] for a concrete counterexample.)

Instead of using $*$, we use $\land$. One might instead conjecture a $\land$-based rule like $(G \twoheadrightarrowtail_{\mathcal{E}} A) * (G \twoheadrightarrowtail_{\mathcal{E}} B) \vdash (G \twoheadrightarrowtail_{\mathcal{E}} A \land B)$; this rule still is not sound on its own (again, see the extended version for a concrete counterexample), but fortunately, it becomes sound as long as we add another point proposition condition (Guard-And). This rule is especially useful in combination with PCM-And, which can be used to deduce the premise of Guard-And.

3.3 Using $\twoheadrightarrowtail$ in a Program Logic

Deriving heap rules. Iris is not a separation logic for a single programming language; rather, it is a general separation logic *framework* which can be used to instantiate a program logic for any user-provided programming language. In other words, rules like the following, which might be considered "primitive" rules within a program logic, can actually be derived soundly within Iris.

Heap-Ref
$$\{\}\ \texttt{ref}(v)\ \{\ell.\ \ell \hookrightarrow v\}$$

Heap-Free
$$\{\ell \hookrightarrow v\}\ \texttt{free}(v)\ \{\}$$

Heap-Write
$$\{\ell \hookrightarrow v\}\ \ell \leftarrow v'\ \{\ell \hookrightarrow v'\}$$

Heap-Read
$$\{\ell \hookrightarrow v\}\ !\ell\ \{r.\ \ell \hookrightarrow v * v = r\}$$

Let us overview this process, and then explain how it works with Leaf's $\twoheadrightarrowtail$ in the picture.

To instantiate a program logic, the user provides their programming language and its operational semantics. Here, we consider heap semantics operating over a state given by $\sigma : \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Value}$, with allocation ($\texttt{ref}$), deallocation ($\texttt{free}$), assignment ($\leftarrow$) and reading ($!$). Next, the user gives meaning to the heap state σ within the separation logic by defining an interpretation of the heap state as a proposition, $\mathcal{H}(\sigma) : \mathit{iProp}$, along with propositions to be manipulated by the user within the program logic (here, $\ell \hookrightarrow v$). Finally, they prove the primitive heap rules via corresponding updates or entailments. For example, the following suffice to show the above four Hoare rules.

$$\begin{aligned}
\mathcal{H}(\sigma) &\Rrightarrow \exists \ell.\, \mathcal{H}(\sigma[\ell := v]) * (\ell \hookrightarrow v') && (\textsc{AllocUpd})\\
\mathcal{H}(\sigma) * (\ell \hookrightarrow v) &\Rrightarrow \mathcal{H}(\sigma \backslash \{\ell\}) && (\textsc{FreeUpd})\\
\mathcal{H}(\sigma) * (\ell \hookrightarrow v) &\vdash (\sigma(\ell) = v) && (\textsc{ReadEq})\\
\mathcal{H}(\sigma) * (\ell \hookrightarrow v) &\Rrightarrow \mathcal{H}(\sigma[\ell := v']) * (\ell \hookrightarrow v') && (\textsc{WriteUpd})
\end{aligned}$$

Thus, it suffices for the user to construct $\mathcal{H}(\sigma)$ and $\ell \hookrightarrow v$ so that the above hold; this can be done via a custom PCM construction, using PCM-Valid to prove ReadEq, PCM-Update to prove WriteUpd and FreeUpd, and PCM-Alloc to prove AllocUpd.

Now, the new rule we want to construct is, for any $\ell, v, \mathcal{E}, \mathcal{E}_1$,

Heap-Read-Shared

$$[\ell \hookrightarrow v]_{\mathcal{E}}\ \{\}\ !\ell\ \{v'.\ v = v'\}_{\mathcal{E}\cup\mathcal{E}_1}$$

Expanding the notation, this is equivalent to, for any $G : \mathit{iProp}$,

$$\{G * (G \twoheadrightarrowtail_{\mathcal{E}} (\ell \hookrightarrow v))\}\ !\ell\ \{v'.\ G * (v = v')\}_{\mathcal{E}\cup\mathcal{E}_1}$$

This follows from,

$$(G \twoheadrightarrowtail_{\mathcal{E}} (\ell \hookrightarrow v)) \vdash \mathcal{H}(\sigma) * G \Rrightarrow_{\mathcal{E}\cup\mathcal{E}_1} \mathcal{H}(\sigma) * G * (\sigma(\ell) = v)$$

and this in turn follows from Unguard-Pers and ReadEq. Notably, we do not need to re-do the construction of $\mathcal{H}(\sigma)$ or $\ell \hookrightarrow v$ to support the derivation of Heap-Read-Shared. Along with the new deduction rules for $\twoheadrightarrowtail$, the old construction "just works."

Atomic Invariants. Propositions shared via guarding can serve as *atomic invariants*; i.e., we can obtain *exclusive* ownership of a shared proposition for the duration of an atomic operation, as long as we restore the invariant at the end of the operation.

Guard-Atomic-Inv

$$\frac{[\cdots]\ \{P * X\}\ e\ \{Q * X\}_{\mathcal{E}_1} \qquad \mathcal{E} \cap \mathcal{E}_1 = \emptyset \qquad e \text{ is atomic}}{[X]_{\mathcal{E}}\ [\cdots]\ \{P\}\ e\ \{Q\}_{\mathcal{E}\cup\mathcal{E}_1}}$$

Non-Atomic Memory. The heap semantics in the preceding example use sequentially consistent, atomic heap operations. But what about other memory ordering models?

We can also apply Leaf to heap semantics that model *non-atomic memory access*, i.e., memory accesses for which data races are entirely disallowed, alongside atomic operations. Non-atomic memory has been modeled before in Iris, e.g., by RustBelt [Jung et al. 2017], which models each non-atomic operation as two execution steps in order to detect overlapping operations. We can apply Leaf to this situation, and prove Heap-Read-Shared for non-atomic reads; however, the proof is slightly more challenging than it is for the purely-atomic heap semantics, primarily because the heap semantics have to model non-atomic reads as effectful operations. To get around this, we need to be slightly clever in our definition of $\mathcal{H}(\sigma)$; see our extended paper [Hance et al. 2023b] for a sketch.

3.4 Storage Protocols

Leaf's *storage protocol* is a formulation of custom ghost state whose unique feature is its laws allowing deductions of nontrivial $\twoheadrightarrowtail$ propositions. Storage protocols are similar to the ghost state presented earlier, which embeds elements of a monoid as separation logic propositions $\boxed{x}^{\gamma}$ and uses a derived relation ($\rightsquigarrow$) to deduce updates ($\Rrightarrow$). The storage protocol formulation, however, is given as a relationship between *two* monoids, a *protocol monoid* P and a *storage monoid* S. Elements $p : P$ are embedded as propositions $\langle p \rangle^{\gamma}$, analogously to $\boxed{x}^{\gamma}$, while elements $s : S$ are embedded via an arbitrary proposition family $F : S \longrightarrow \mathit{iProp}$, which is specified upon initialization of the ghost

A **storage protocol** consists of:

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

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

A *protocol monoid*, that is, a (total) commutative monoid $(P, \cdot)$, with an arbitrary predicate $C : P \longrightarrow \mathsf{Bool}$ and function $\mathcal{S} : P|_C \longrightarrow S$ (i.e., the domain of $\mathcal{S}$ is restricted to the subset of P where C holds) where,

$$\forall a.\ a \cdot \epsilon = a$$
$$\forall a, b.\ a \cdot b = b \cdot a$$
$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c)$$
$$\forall a.\ C(a) \Rightarrow \mathcal{V}(\mathcal{S}(a))$$

Note that C (unlike $\mathcal{V}$) is *not* necessarily closed under $\preceq$.

Derived relations for storage protocols:

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

$$\begin{aligned}
(p, s) \overset{\cdot\ \cdot}{\rightsquigarrow} (p', s') &\triangleq \forall q.\ C(p \cdot q) \Rightarrow (C(p' \cdot q) \\
&\quad \land \mathcal{V}(\mathcal{S}(p \cdot q) \cdot s) \\
&\quad \land \mathcal{S}(p \cdot q) \cdot s = \mathcal{S}(p' \cdot q) \cdot s') \\
p \overset{\ \ \cdot}{\rightsquigarrow} (p', s') &\triangleq (p, \epsilon) \overset{\cdot\ \cdot}{\rightsquigarrow} (p', s') \\
(p, s) \overset{\cdot\ \ }{\rightsquigarrow} p' &\triangleq (p, s) \overset{\cdot\ \cdot}{\rightsquigarrow} (p', \epsilon) \\
p \rightsquigarrow p' &\triangleq (p, \epsilon) \overset{\cdot\ \cdot}{\rightsquigarrow} (p', \epsilon) \\
p \nrightarrow s &\triangleq \forall q.\ C(p \cdot q) \Rightarrow s \preceq \mathcal{S}(p \cdot q)
\end{aligned}$$

(a) Definitions.

Storage Protocol Logic

Instantiated for a given storage protocol $(S, \cdot, \mathcal{V}), (P, \cdot), C, \mathcal{S}$

Propositions: $\langle p \rangle^\gamma$

Persistent propositions: $\mathsf{sto}(\gamma, F)$

(where $\gamma : \mathit{Name},\ F : S \longrightarrow \mathit{iProp},\ p : P$)

$$\begin{aligned}
\mathsf{RespectsComposition}(F) &\triangleq (F(\epsilon) \dashv\vdash \mathsf{True}) \\
\text{and } \forall x, y.\ \mathcal{V}(x \cdot y) &\Rightarrow (F(x \cdot y) \dashv\vdash F(x) * F(y))
\end{aligned}$$

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

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

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

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

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

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

SP-Guard
$$\frac{p \nrightarrow s}{\mathsf{sto}(\gamma, F) \vdash \langle p \rangle^\gamma \rightarrowtail_\gamma (\triangleright F(s))}$$

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

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

SP-Valid
$$\langle p \rangle^\gamma \vdash \exists q.\ C(p \cdot q)$$

(b) Deduction rules.

Fig. 5. Storage protocols and derived relations.

state (SP-Alloc). The update relation and the resulting $\Rrightarrow$ propositions are complicated by the need to account for S, and we also include a new derived relation, $\nrightarrow$, from which we deduce guards of the form $\langle p \rangle^\gamma \rightarrowtail_{\mathcal{E}} F(s)$. Figure 5a gives the precise definitions for these derived relations.

Let us first discuss the role of S, and its impact on our definition of updates. P and S are related by a *storage function* $\mathcal{S} : P \longrightarrow S$, which intuitively associates to each possible state of P some value $s : S$ that is "stored," and since the stored value might change upon any update, we need to account for this in the definition of updates. For example, when the P-state updates, the corresponding stored state $(\mathcal{S}(p))$ might also change; if the stored state changes from t to $t \cdot s$, we consider s to be "deposited"; if the state changes the other way, we consider it withdrawn. The most general form of an exchange is given by $\overset{\cdot\ \cdot}{\rightsquigarrow}$, with the other three as special cases: A withdraw ($\overset{\ \ \cdot}{\rightsquigarrow}$) occurs when "nothing" (i.e., the unit) is deposited, a deposit ($\overset{\cdot\ \ }{\rightsquigarrow}$) occurs when nothing is withdrawn, and a "normal" update ($\rightsquigarrow$) occurs when the stored value remains constant. These relations then give rise to updates in the separation logic (SP-Exchange, and its special cases, SP-Deposit, SP-Withdraw, SP-Update), operating on $\langle p\rangle^\gamma$ and $F(s)$. Specifically, a deposit of s allows an update that gives up ownership of $F(s)$, while a withdraw of s allows an update that obtains ownership of $F(s)$.

Now, with the ability to "deposit" elements into the protocol and "withdraw" them, we can add the ability to have shared access to those stored elements: the derived relation $p \mathrel{⇸} s$ gives rise to $\langle p\rangle^\gamma \mathrel{⤖} F(s)$ by SP-Guard. Let us unpack the definition of $\mathrel{⇸}$ to understand intuitively why this should work: $p \mathrel{⇸} s$ is defined as $\forall q.\ C(p\cdot q) \Rightarrow s \preceq \mathcal{S}(p\cdot q)$; this essentially says that p is a "witness" that the value s is stored; i.e., if we have ownership of p, then any "completion" of the state $p\cdot q$, which is valid according to the validity function C, must be storing something $\succeq s$.

Initialization of a protocol. When initializing a protocol (SP-Alloc) we get to specify F, and we also get to specify the initial element $s : S$ while giving up $F(s)$, the initial proposition to be stored. Inheriting yet another trick from Iris, we can also specify a *namespace* $\mathcal{N}$, a subset of *Name*, that the resulting protocol name has to be in. Using fixed namespaces (such as $\mathcal{F}rac$ or *Count* from Figure 2) is often more convenient than managing individual names γ that cannot be known *a priori*.

Example 3.2 (Fractional protocol for a single proposition). To start simple, let us suppose we have a single proposition, Q, we would like to manage. Set the protocol monoid $P \triangleq \mathbb{Q}_{\geq 0}$ and the storage monoid $S \triangleq \mathbb{N}$, with composition as addition in both cases and a unit of 0. Set C to be true exactly on the integers, and for integers n, set $\mathcal{S}(n) \triangleq n$. Let $\mathcal{V}$ always be true. Now, we have the exchange $(1,0) \overset{\cdot\,\cdot}{\rightsquigarrow} (0,1)$ (also written as a withdraw, $1 \overset{\cdot}{\rightsquigarrow} (0,1)$) and the reverse, $(0,1) \overset{\cdot\,\cdot}{\rightsquigarrow} (1,0)$ (also written as a deposit, $(0,1) \overset{\cdot}{\rightsquigarrow} 1$). Finally, for any $q > 0$, we have $q \mathrel{⇸} 1$. This is the key property that says the fraction q can act as a read-only element, and it follows from the following argument: if $q' \geq q$ and $C(q')$ holds, then q' is an integer and $\mathcal{S}(q') = q' \geq 1$.

Finally, set the proposition family $F(n) \triangleq \mathop{\Large\ast}^{n} Q$, i.e., Q conjoined n times. Now we can say that,

$$
\begin{array}{lll}
\text{True} \Rrightarrow \exists\gamma.\ \mathsf{sto}(\gamma, F) & & \text{(via SP-Alloc)} \\
\mathsf{sto}(\gamma, F) \vdash \triangleright Q \Rrightarrow_\gamma \langle 1\rangle^\gamma & & \text{(via SP-Deposit)} \\
\mathsf{sto}(\gamma, F) \vdash \langle 1\rangle^\gamma \Rrightarrow_\gamma \triangleright Q & & \text{(via SP-Withdraw)} \\
\mathsf{sto}(\gamma, F) \vdash \langle q\rangle^\gamma \mathrel{⤖}_\gamma \triangleright Q & (\text{for } q > 0) & \text{(via SP-Guard)}
\end{array}
$$

Example 3.3 (Fractional memory permissions). Assume points-to propositions $\ell \hookrightarrow v$ are given. We wish to construct $\ell \xhookrightarrow{\mathsf{frac}}_q v$ and the laws as given in Figure 2.

We take $P = (Loc \times Value) \xrightarrow{\mathsf{fin}} \mathbb{Q}_{\geq 0}$ and $S = (Loc \times Value) \xrightarrow{\mathsf{fin}} \mathbb{N}$, defining $\cdot, \mathcal{V}, C, \mathcal{S}$ elementwise using the definitions of the previous example. Define F such that $F([(\ell, v) \mapsto 1]) = \ell \hookrightarrow v$. Instantiate the protocol to obtain a location $\gamma \in \mathcal{F}rac$, and then set $\ell \xhookrightarrow{\mathsf{frac}}_q v \triangleq \langle[(\ell, v) \mapsto q]\rangle^\gamma$. From here we can derive the appropriate withdraw, deposit, and guard.

The counting protocol (Figure 2) can be done similarly. (See the extended version of our paper [Hance et al. 2023b] for details.)

Example 3.4 (Forever Protocol). The most basic sharing pattern is to make something freely shareable forever (analogous to Iris invariants $\boxed{Q}$). We can express this succinctly by guarding Q with True: $Q \Rrightarrow (\text{True} \mathrel{⤖}_{\mathcal{F}orever} \triangleright Q)$. To derive this, we use a storage protocol: let the protocol monoid P be the trivial monoid $\{\epsilon\}$ and the storage monoid S be $\textsc{Excl}(\mathbf{1})$, with $\mathcal{S}(\epsilon) \triangleq \mathsf{ex}(1)$. This protocol has no interesting updates; it can only be initialized. Let $F(\mathsf{ex}(1)) \triangleq Q$. By SP-Alloc we have $Q \Rrightarrow \exists\gamma.\ \mathsf{sto}(\gamma, F) * \langle\epsilon\rangle^\gamma * (\gamma \in \mathcal{F}orever)$. Now we can chain True $\mathrel{⤖} \langle\epsilon\rangle^\gamma$ (by Guard-Pers) and $\langle\epsilon\rangle^\gamma \mathrel{⤖}_{\mathcal{F}orever} (\triangleright Q)$ (by SP-Guard).

3.5 Handling the Later Modality $\triangleright$

Note that some rules in Figure 5b use the *later modality*, $\triangleright$, a feature of step-indexed logics like Iris. In Leaf, $\triangleright$ allows us to dynamically specify the proposition families F during protocol initialization

```
rwlock_new() ≜ {exc : ref(False), rc : ref(0)}

lock_exc(rw) ≜
  do
    let success = CAS(rw.exc, False, True)
  until success
  do
    let r = !rw.rc
  until r = 0

unlock_exc(rw) ≜ rw.exc ⟵ 0
```

```
rwlock_free(rw) ≜ free(rw.exc); free(rw.rc)

lock_shared(rw) ≜
  do
    FetchAdd(rw.rc, 1);
    let exc = !rw.exc in
    if exc then FetchAdd(rw.rc, −1);
  until exc = False

unlock_shared(rw) ≜ FetchAdd(rw.rc, −1)
```

Fig. 6. Implementation of a reader-writer lock. We assume all heap operations are atomic, including CAS (compare-and-swap) and FetchAdd.

(SP-Alloc); without ▷, this would be unsound. If we gave up that ability and instead specified all families *a priori*, we could remove ▷ from the rest of the rules. (This is analogous to Iris requiring ▷ for dynamically allocated invariants.) Leaf provides rules to eliminate ▷ from within guards:

$$\frac{\text{timeless}(P)}{(\triangleright P) ⤀_{\mathcal{E}} P}\ (\textsc{Later-Guard}) \qquad \frac{\text{timeless}(P) \quad \text{persistent}(Q)}{G * (G ⤀_{\mathcal{E}} \triangleright(P * Q)) \Rrightarrow_{\mathcal{E}} G * \triangleright(G ⤀_{\mathcal{E}} (P * Q))}\ (\textsc{Later-Pers-Guard})$$

Timelessness [Krebbers et al. 2017] is a technical condition that effectively says a proposition is "independent of the step-index," which makes it easier to account for ▷ modalities. Timelessness holds for both the PCM-based $\boxed{x}^{\gamma}$ and our $\langle x \rangle^{\gamma}$ propositions. (Technically, this is because PCMs are special cases of "discrete CMRAs.") However, $\text{sto}(\gamma, F)$ is *not* timeless, since it depends recursively on *iProp*, but since it *is* persistent, we can use Later-Pers-Guard for such propositions.

4 RWLOCK EXAMPLE: VERIFYING A CUSTOM PROTOCOL FOR SHARING STATE

Our two case studies are arranged to show the two "halves" of Leaf: the first one (this section) demonstrates the verification of a sharing protocol that lets the client acquire shared state, while our second one (§5) shows how a client can make use of the shared state.

Specifically, in this section, we verify a reader-writer lock, one which is slightly more complicated than that which is captured directly by a standard permission logic, a situation which the storage protocol was designed for. The implementation of our reader-writer lock in shown in Figure 6, with an example execution trace in Figure 7. The implementation's main complication here is the fact that acquiring a lock is a two-step process: a thread might increment the reference counter, but then fail to acquire the lock in the second step. Hence, the physical value of the reference counter may not match the number of extant read-references.

Initially, this design might seem strange—why not just put all the data in a single atomic field to simplify the design? However, the use of distinct fields is an essential element of more complex lock designs, such as the multi-counter design mentioned in the introduction, where each counter goes on a different cache line. In fact, simply having an intermediate state at all captures most of the complexity of the multi-counter design, so we use the single-counter implementation for this paper. In the extended paper [Hance et al. 2023b], however, we spell out more details for a multi-counter design.

Proof Overview. We tackle the proof in two stages: first, we devise some useful ghost resources; then, we use those resources in the program logic to prove the implementation meets the specification (Figure 1). The key is to find the right resources and their relationships that we need.

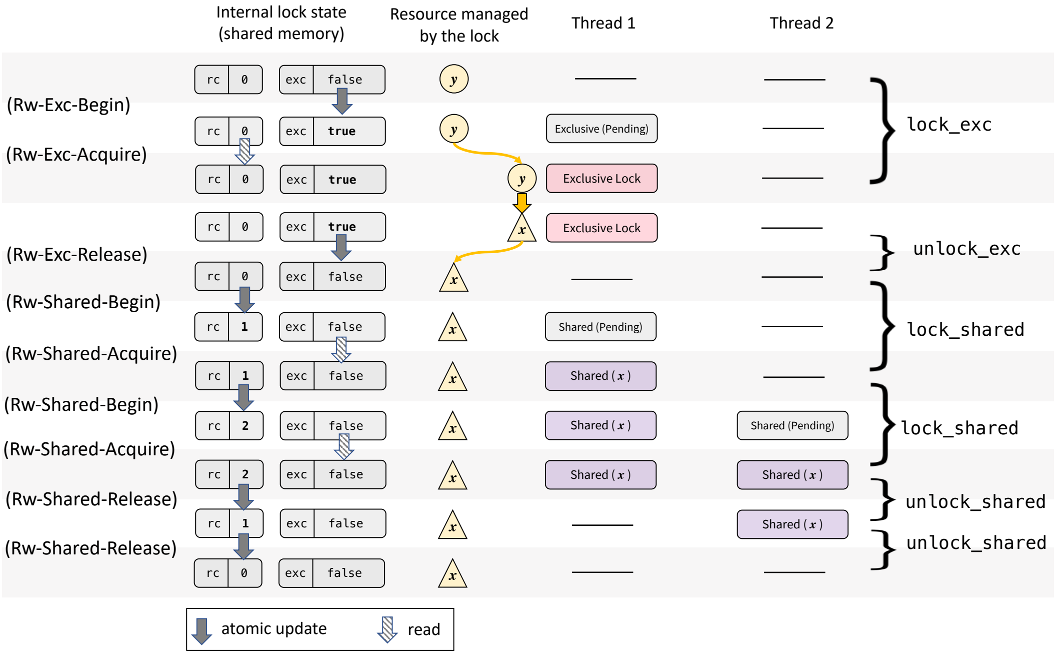

Fig. 7. Example execution of two threads using a shared reader-writer lock. This is an "ideal" execution, without contention or retries. First, we see Thread 1 acquire exclusive lock. This gives them exclusive control over the resource, which they can therefore modify (here, changing it from y to x) before releasing the lock. We then see both threads acquire a shared lock, where they simultaneously have read access to the x resource. On the left, we annotate each step with the ghost resource update from Figure 8 it corresponds to.

The RwLock specification (Figure 1) already indicates three propositions that we need to construct in one way or another: $\mathsf{IsRwLock}(rw, \gamma, F)$, $\mathsf{Exc}(\gamma)$, and $\mathsf{Sh}(\gamma, x)$. We also have $\mathsf{Sh}(\gamma, x) \twoheadrightarrow_\gamma F(x)$ as a desired property. This gives us a reason to use a storage protocol: it allows us to construct new resources and derive $\twoheadrightarrow$ relationships. Since the storage protocol and its resulting resources are more elaborate than in the previous examples, we will take the time here to explain exactly how to come up with the protocol.

First, we naturally need a component to represent the lock's internal state, which we call $\mathsf{Fields}(\gamma, exc, rc, x)$, containing both the exc and rc fields, and also the stored value, x. We can tie the first two fields to the physical, in-memory values with a proposition like the following:

$$\mathsf{IsRwLock}(rw, \gamma, F) \triangleq \exists exc, rc, x.\ \mathsf{Fields}(\gamma, exc, rc, x) * (rw.exc \hookrightarrow exc) * (rw.rc \hookrightarrow rc) * \ldots$$

Next, we use resources to represent the intermediate states that occur during lock acquisition. For example, write-lock acquisition has a moment where we have set exc but not observed rc; likewise, read-lock acquisition has a temporary state where we have incremented rc but not observed exc. We use $\mathsf{ExcPending}(\gamma)$ and $\mathsf{ShPending}(\gamma)$ to represent these states.

So for example, to prove lock_exc, which has the intended specification,

$$[\mathsf{IsRwLock}(rw, \gamma, F)]\ \{\}\ \mathsf{lock\_exc}(rw)\ \{\mathsf{Exc}(\gamma) * \exists x.\ \triangleright F(x)\}$$

its proof outline should look something like,

$$[\exists exc, rc, x.\ \mathsf{Fields}(\gamma, exc, rc, x) * (rw.exc \hookrightarrow exc) * (rw.rc \hookrightarrow rc)]$$

```
{ }
do
  let $success$ = CAS($rw.exc$, False, True)
until $success$
{$\mathsf{ExcPending}(\gamma)$}
do
  let $r$ = !$rw.rc$
until $r = 0$
{$\exists x.\ \mathsf{Exc}(\gamma) * \triangleright F(x)$}
```

With shared access to $rw.exc \hookrightarrow exc$ and $rw.rc \hookrightarrow rc$, we can use Guard-Atomic-Inv to perform the requisite atomic CAS and atomic load. However, because all these resources are shared, we cannot hold onto them for the duration spanning both operations at once. Therefore, when performing the CAS, the triple (exc, rc, x) used might be different than the triple used for the later load instruction. This is why we cannot track the intermediate "pending" state as part of the Fields resource, and need to use a separate ExcPending resource for the thread.

With this sketch in place, we can observe some of the operations we need: for the CAS operation, we update exc from False to True and should somehow obtain $\mathsf{ExcPending}(\gamma)$ in the process. So we need:

$$\mathsf{Fields}(\gamma, \mathsf{False}, rc, x) \Rrightarrow \mathsf{Fields}(\gamma, \mathsf{True}, rc, x) * \mathsf{ExcPending}(\gamma)$$

For the second step, reading the rc value, we find we need:

$$\mathsf{Fields}(\gamma, exc, 0, x) * \mathsf{ExcPending}(\gamma) \Rrightarrow \mathsf{Fields}(\gamma, exc, 0, x) * \mathsf{Exc}(\gamma) * \triangleright F(x)$$

This update requires us to observe that $rc = 0$, though it does not change rc or any of the other fields. It does, however, move us from the pending-exclusive state to the actual exclusive-lock state, while also acquiring exclusive ownership of the protected resource, as was our goal.

Figure 8 shows all of the operations that we need, including those we could determine from a similar analysis of the lock_shared implementation. This also includes an additional proposition, $\mathsf{RwFamily}(\gamma, F)$, that ties γ to the proposition family F.

Now, we just need to use a storage protocol to construct the resources of Figure 8 and prove these the desired updates. Then we can complete the Hoare proofs based on the above plan.

Step 1: Constructing the ghost resources via a storage protocol. The first step in building a storage protocol is to determine the storage monoid and the protocol monoid. In our example, the storage monoid S can be $\textsc{Excl}(X)$; i.e., there is either one thing stored, or there is not.

Our primary effort, then, is the protocol monoid P. We define P to have a component for each class of proposition it needs to support. First up is the Fields proposition, and we know there should always be one such; therefore, we can represent it with Excl. Next, there should always be at most one of ExcPending or Exc, so we can use Excl for these as well. Meanwhile, there might be any number of ShPending propositions at a given time, so we can use $\mathbb{N}$ for these.

RwLock Storage Protocol Resource

Propositions: $\mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rc}, x) \quad \mathsf{ExcPending}(\gamma) \quad \mathsf{Exc}(\gamma) \quad \mathsf{ShPending}(\gamma) \quad \mathsf{Sh}(\gamma, x)$

Persistent Propositions: $\mathsf{RwFamily}(\gamma, F)$

(where $\gamma : \mathit{Name},\ \mathit{exc} : \mathsf{Bool},\ \mathit{rc} : \mathbb{Z},\ X : \mathit{Set},\ x : X,\ F : X \to \mathit{iProp}$)

$$F(x) \Rrightarrow \exists \gamma.\ \mathsf{Fields}(\gamma, \mathsf{False}, 0, x) * \mathsf{RwFamily}(\gamma, F) \qquad (\textsc{Rw-Init})$$

$$\begin{aligned}
&\mathsf{RwFamily}(\gamma, F) \vdash \\
&\mathsf{Fields}(\gamma, \mathsf{False}, \mathit{rc}, x) \Rrightarrow_{\gamma} \mathsf{Fields}(\gamma, \mathsf{True}, \mathit{rc}, x) * \mathsf{ExcPending}(\gamma) && (\textsc{Rw-Exc-Begin})\\
&\mathsf{Fields}(\gamma, \mathit{exc}, 0, x) * \mathsf{ExcPending}(\gamma) \Rrightarrow_{\gamma} \mathsf{Fields}(\gamma, \mathit{exc}, 0, x) * \mathsf{Exc}(\gamma) * \triangleright F(x) && (\textsc{Rw-Exc-Acquire})\\
&\mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rc}, y) * \mathsf{Exc}(\gamma) * \triangleright F(x) \Rrightarrow_{\gamma} \mathsf{Fields}(\gamma, \mathsf{False}, \mathit{rc}, x) && (\textsc{Rw-Exc-Release})\\
&\mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rc}, x) \Rrightarrow_{\gamma} \mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rc} + 1, x) * \mathsf{ShPending}(\gamma) && (\textsc{Rw-Shared-Begin})\\
&\mathsf{Fields}(\gamma, \mathsf{False}, \mathit{rc}, x) * \mathsf{ShPending}(\gamma) \Rrightarrow_{\gamma} \mathsf{Fields}(\gamma, \mathsf{False}, \mathit{rc}, x) * \mathsf{Sh}(\gamma, x) && (\textsc{Rw-Shared-Acquire})\\
&\mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rc}, x) * \mathsf{Sh}(\gamma, y) \Rrightarrow_{\gamma} \mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rc} - 1, x) && (\textsc{Rw-Shared-Release})\\
&\mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rc}, x) * \mathsf{ShPending}(\gamma) \Rrightarrow_{\gamma} \mathsf{Fields}(\gamma, \mathit{exc}, \mathit{rc} - 1, x) && (\textsc{Rw-Shared-Retry})\\
&\mathsf{Sh}(\gamma, x) \mathrel{⤔}_{\gamma} \triangleright F(x) && (\textsc{Rw-Shared-Guard})
\end{aligned}$$

Fig. 8. A custom resource derived via the storage protocol mechanism, designed for our particular implementation.

Finally, for the Sh propositions, we can have any number, but they need to agree on the value of x. For this, we use a monoid $\textsc{AgN}(X)$, which tracks a single value and a count. It is given by,

$$\epsilon \mid \mathsf{agn}(x, n) \mid \lightning \qquad \text{where } x : X \text{ and } n : \mathbb{N},\ n \geq 1$$
$$\text{with } \mathsf{agn}(x, n) \cdot \mathsf{agn}(x, m) = \mathsf{agn}(x, n + m) \text{ and for } x \neq y,\ \mathsf{agn}(x, n) \cdot \mathsf{agn}(y, m) = \lightning$$

All in all, we can now declare our protocol monoid and name its important elements:

$$P \triangleq \textsc{Excl}(\mathsf{Bool} \times \mathbb{Z} \times X) \times \textsc{Excl}(\mathbf{1}) \times \textsc{Excl}(\mathbf{1}) \times \mathbb{N} \times \textsc{AgN}(X)$$

$$\begin{array}{rlllll}
\mathsf{fields}(\mathit{exc}, \mathit{rc}, x) & \triangleq (\mathsf{ex}((\mathit{exc}, \mathit{rc}, x)), & \epsilon, & \epsilon, & 0, & \epsilon)\\
\mathsf{excPending} & \triangleq (\epsilon, & \mathsf{ex}, & \epsilon, & 0, & \epsilon)\\
\mathsf{exc} & \triangleq (\epsilon, & \epsilon, & \mathsf{ex}, & 0, & \epsilon)\\
\mathsf{shPending} & \triangleq (\epsilon, & \epsilon, & \epsilon, & 1, & \epsilon)\\
\mathsf{sh}(x) & \triangleq (\epsilon, & \epsilon, & \epsilon, & 0, & \mathsf{agn}(x, 1))
\end{array}$$

Now, we need to define $\mathcal{S}$ and C. First, $\mathcal{S}$ determines the element stored; this is given by x in the fields state, unless the lock is currently exclusively taken, in which case the storage is empty:

$$\mathcal{S}(\mathsf{ex}((\mathit{exc}, \mathit{rc}, x)), \_, \epsilon, \_, \_) \triangleq \mathsf{ex}(x) \qquad \mathcal{S}(\mathsf{ex}((\mathit{exc}, \mathit{rc}, x)), \_, \mathsf{ex}, \_, \_) \triangleq \epsilon$$

Next, we define C to be False if any entry is $\lightning$ or the first entry is ϵ; otherwise,

$$\begin{aligned}
&C((\mathsf{ex}((\mathit{exc}, \mathit{rc}, x)), \mathit{ep}, e, \mathit{sp}, s)) \ \triangleq\ (\mathit{rc} = \mathit{sp} + \mathit{count}(s)) \land (\neg \mathit{exc} \Rightarrow \mathit{ep} = \epsilon \land e = \epsilon)\\
&\quad \land (\mathit{exc} \Rightarrow (\mathit{ep} = \mathsf{ex} \lor e = \mathsf{ex}) \land \neg(\mathit{ep} = \mathsf{ex} \land e = \mathsf{ex})) \land (e = \mathsf{ex} \Rightarrow s = \epsilon) \land (\forall y, n.\ s = \mathsf{agn}(y, n) \Rightarrow x = y)\\
&(\text{where } \mathit{count}(\mathsf{agn}(x, n)) = n \text{ and } \mathit{count}(\epsilon) = 0)
\end{aligned}$$

These predicates can be stated in plain English: the reference count *rc* is the total number of threads with the shared lock or in the process of acquiring it; the *exc* field indicates whether any thread has the exclusive lock or is in the process of acquiring it; an exclusive lock cannot be taken at the same time as a shared lock; the value taken by a shared lock should match the fields's x value.

Now, with the storage protocol established, we can embed these elements as propositions: we let $\mathsf{Fields}(\gamma, exc, rc, x) \triangleq \langle \mathsf{fields}(exc, rc, x) \rangle^{\gamma}$ and so on. We also let $\mathsf{RwFamily}(\gamma, F) \triangleq \mathsf{sto}(\gamma, F')$, where $F'(\mathsf{ex}(x)) \triangleq F(x)$ and $F'(\epsilon) \triangleq \mathsf{True}$. Now, in order to show our desired reader-writer lock rules (Figure 8), it suffices to show the following updates (by SP-Update):

$$\begin{aligned}
\mathsf{fields}(\mathsf{False}, rc, x) &\rightsquigarrow \mathsf{fields}(\mathsf{True}, rc, x) \cdot \mathsf{excPending} \\
\mathsf{fields}(exc, rc, x) &\rightsquigarrow \mathsf{fields}(exc, rc + 1, x) \cdot \mathsf{shPending} \\
\mathsf{fields}(\mathsf{False}, rc, x) \cdot \mathsf{shPending} &\rightsquigarrow \mathsf{fields}(\mathsf{False}, rc, x) \cdot \mathsf{sh}(x) \\
\mathsf{fields}(exc, rc, x) \cdot \mathsf{sh}(y) &\rightsquigarrow \mathsf{fields}(exc, rc - 1, x) \\
\mathsf{fields}(exc, rc, x) \cdot \mathsf{shPending} &\rightsquigarrow \mathsf{fields}(exc, rc - 1, x)
\end{aligned}$$

As well as a withdraw (by SP-Withdraw) and a deposit (by SP-Deposit):

$$\begin{aligned}
\mathsf{fields}(exc, 0, x) \cdot \mathsf{excPending} &\overset{\;\;\;\cdot}{\rightsquigarrow} (\mathsf{fields}(exc, 0, x) \cdot \mathsf{exc}, \mathsf{ex}(x)) \\
(\mathsf{fields}(exc, rc, y) \cdot \mathsf{exc}, \mathsf{ex}(x)) &\overset{\cdot\;\;\;}{\rightsquigarrow} \mathsf{fields}(\mathsf{False}, rc, x)
\end{aligned}$$

And finally, a guard (by SP-Guard):

$$\mathsf{sh}(x) \mathrel{⇸} \mathsf{ex}(x)$$

Finally, we can prove all these just by expanding the definitions and using the logical invariants encoded in $\mathcal{C}$.

Let us summarize exactly what the storage protocol gave us in this particular proof strategy. We wanted to construct some set of ghost resources with certain relationships, mostly updates ($\Rrightarrow$), representing specific implementation details of the lock, with a single $⤔$ proposition that enables sharing. The storage protocol shows how to reduce those desired relationships to proof obligations ($\rightsquigarrow$, $\overset{\;\;\;\cdot}{\rightsquigarrow}$, $\overset{\cdot\;\;\;}{\rightsquigarrow}$, $⇸$) about monoids that can be expressed in first-order logic. These obligations all encode properties that should map cleanly to an intuitive property of the system, e.g., the $\overset{\;\;\;\cdot}{\rightsquigarrow}$ proposition intuitively means "from the intermediate pending state, if $rc = 0$, then the stored resource can be withdrawn," while $\mathsf{sh}(x) \mathrel{⇸} \mathsf{ex}(x)$ intuitively means "any reader agrees with the source-of-truth on what the shared value is." These properties all rely on our definition of $\mathcal{S}$, a predicate that encodes which states of the system are well-formed.

Step 2: Verifying the implementation. To verify the implementation (Figure 6) against the spec (Figure 1), we first need to nail down a definition for $\mathsf{IsRwLock}(rw, \gamma, F)$. Since γ is meant to be the unique identifier for the reader-writer lock, we can have it be the same as the ghost name γ from the RwLock logic, and likewise F, the family of propositions protected in the lock, be the same as F, the family of propositions protected by the RwLock protocol.

The propositions representing the reader-writer lock should, as a whole, include the proposition $\mathsf{RwFamily}(\gamma, F)$, the permission to access the *rw.exc* and *rw.rc* memory cells, and the Fields proposition which has the ghost data to match the contents of the memory cells.

$$\mathsf{IsRwLock}(rw, \gamma, F) \triangleq \mathsf{RwFamily}(\gamma, F) * \exists exc, rc, x.\ \mathsf{Fields}(\gamma, exc, rc, x) * (rw.exc \hookrightarrow exc) * (rw.rc \hookrightarrow rc)$$

The proof for rwlock_new is then straightforward: the implementation allocates the *rw.exc* and *rw.rc* memory, and we can ghostily instantiate the RwLock protocol via Rw-Init. Likewise, the proof for rwlock_free is straightforward, since its precondition requires that the caller has exclusive access to IsRwLock, so we can destructure it and use the exclusive $\hookrightarrow$ propositions in order to call `free`.

The proofs for the other methods, which must operate over a *shared* IsRwLock, are more interesting. Recall the proof outline for lock_exc:

```
$[\mathsf{RwFamily}(\gamma, F) * \exists exc, rc, x.\ \mathsf{Fields}(\gamma, exc, rc, x) * (rw.exc \hookrightarrow exc) * (rw.rc \hookrightarrow rc)]$
  $\{\ \}$
  do
    let $success$ = CAS($rw.exc$, False, True)          (Rw-Exc-Begin)
  until $success$
  $\{\mathsf{ExcPending}(\gamma)\}$
  do
    let $r$ = $!rw.rc$                                  (Rw-Exc-Acquire)
  until $r = 0$
  $\{\exists x.\ \mathsf{Exc}(\gamma) * \triangleright F(x)\}$
```

The gist is that, in the first half, we apply Rw-Exc-Begin (in the case that the CAS succeeds) to obtain $\mathsf{ExcPending}(\gamma)$, and in the second half, we apply Rw-Exc-Acquire to complete the acquisition and obtain the desired state $\mathsf{Exc}(\gamma) * \triangleright F(x)$.

Let us walk through the first half in detail. Since CAS is atomic, we can apply Guard-Atomic-Inv to "open" the shared proposition for the duration of the atomic operation. Thus, we need to show,

$$\{\mathsf{RwFamily}(\gamma, F) * \exists exc, rc, x.\ \mathsf{Fields}(\gamma, exc, rc, x) * (rw.exc \hookrightarrow exc) * (rw.rc \hookrightarrow rc)\}$$

$$\mathtt{CAS}(rw.exc, \mathsf{False}, \mathsf{True})$$

$$\{success.\ ((success = \mathsf{True} * \mathsf{ExcPending}(\gamma)) \lor (success = \mathsf{False})) *$$
$$\mathsf{RwFamily}(\gamma, F) * \exists exc, rc, x.\ \mathsf{Fields}(\gamma, exc, rc, x) * (rw.exc \hookrightarrow exc) * (rw.rc \hookrightarrow rc)\}$$

If CAS succeeds, we have $exc = \mathsf{False}$, so we apply Rw-Exc-Begin. This ensures we have $\mathsf{ExcPending}(\gamma)$ in the $success = \mathsf{True}$ case. Otherwise, we do nothing, and the program loops. The second half of lock_exc, where we atomically read $rw.rc$, is the same, using Rw-Exc-Acquire.

We can use a similar outline for lock_shared:

```
$[\mathsf{RwFamily}(\gamma, F) * \exists exc, rc, x.\ \mathsf{Fields}(\gamma, exc, rc, x) * (rw.exc \hookrightarrow exc) * (rw.rc \hookrightarrow rc)]$
  $\{\ \}$
  do
    $\{\ \}$
    FetchAdd($rw.rc$, 1);                                             (Rw-Shared-Begin)
    $\{\mathsf{ShPending}(\gamma)\}$
    let $exc$ = $!rw.exc$ in                                          (Rw-Shared-Acquire)
    $\{(exc = \mathsf{False} * \exists x.\ \mathsf{Sh}(\gamma, x)) \lor (exc = \mathsf{True} * \mathsf{ShPending}(\gamma))\}$
    if $exc$ then FetchAdd($rw.rc$, $-1$);                            (Rw-Shared-Retry)
    $\{(exc = \mathsf{False} * \exists x.\ \mathsf{Sh}(\gamma, x)) \lor (exc = \mathsf{True})\}$
  until $exc = \mathsf{False}$
  $\{\exists x.\ \mathsf{Sh}(\gamma, x)\}$
```

Finally, the proofs for lock_shared, and unlock_shared all follow similarly, using Rw-Exc-Release and Rw-Shared-Release respectively.

5 HASH TABLE EXAMPLE: COMPOSITION WITH SHARED STATE

In this section, we show how to use Leaf to verify a larger application built on top of the reader-writer lock developed in the previous section. In particular, while the reader-writer lock gives us a mechanism to *obtain* and *release* shared ghost state, here we will see how to *make use* of shared state. In particular, we show we can employ a fine-grained locking scheme, acquire multiple shared locks, and compose state to perform nontrivial operations. While the previous section was primarily an application of the storage protocol (§3.4), this section will primarily be about applying the general-purpose Leaf rules and ghost state (§3.1-§3.3).

The example we consider is a concurrent *linear probing hash table* [Knuth 1998], using a single lock per entry of the hash table. We choose this example primarily because a single operation requires us to take multiple locks, and in particular, for a query operation, we will be taking those locks in shared, read-only mode. Thus, this example tests Leaf's ability as a logic for manipulating shared resources.

Linear probing, here, is a particular strategy a hash table uses to handle hash collisions. Specifically, the hash table is arranged as an array with indices $0 \leq i < L$, with some hash function $H : Key \longrightarrow [0, L)$. To insert a key-value pair (k, v), we attempt to insert it into the array at position $i = H(k)$. If there is a different key already in the slot i, then we attempt to insert into $i + 1$, and so on. Queries are similar: we scan starting at i until we find the key or an empty slot.

In our implementation, we let ht be a record $\{slots, locks\}$ consisting of two arrays: one for the hash table slots, and one for the locks protecting them. We then implement query and update.

```
query(ht, k) ≜ query_iter(ht, k, H(k))
query_iter ≜ rec query_iter(ht, k, i).
    if i ≥ L then abort else
        lock_shared(ht.locks[i]);
        let r = (match !ht.slots[i] with
            | None ⇒ None
            | Some((k_i, v_i)) ⇒
                if k = k_i then Some(v_i)
                else query_iter(ht, k, i + 1)
        end) in
        unlock_shared(ht.locks[i]);
        r
```

```
update(ht, k) ≜ update_iter(ht, k, H(k))
update_iter ≜ rec update_iter(ht, k, i).
    if i ≥ L then abort else
        lock_exc(ht.locks[i]);
        match !ht.slots[i] with
            | None ⇒ ht.slots[i] ← Some((k, v))
            | Some((k_i, v_i)) ⇒
                if k = k_i then
                    ht.slots[i] ← Some((k, v))
                else update_iter(ht, k, i + 1)
        end
        unlock_exc(ht.locks[i]);
```

For simplicity, we do not handle hash table resizing (i.e., we assume a fixed L). For the paper version only, the program aborts whenever a probe reaches the end of the table (index L). This keeps the presentation manageable, though our full Coq version does gracefully handle the last case.

First, let us nail down the spec we want to prove. There are a handful of options; here, we choose one that manages the key-value mapping with propositions $m(\gamma_{\text{ht}}, k, v)$, (analogous to $\hookrightarrow$ propositions). As with the reader-writer lock, we also have a main proposition $\mathsf{IsHT}(ht, \gamma_{\text{ht}})$ to say that ht is a hash table, and since the hash table is concurrent, the specifications for query and update require a shared $[\mathsf{IsHT}(ht, \gamma_{\text{ht}})]$.

Hash Table Specification

Propositions: $\mathsf{IsHT}(ht, \gamma_{\text{ht}})$ $\quad m(\gamma, k, v)$ $\qquad$ (where $ht : Value,\ \gamma_{\text{ht}} : Name,\ k : Key,\ v : Value^?$)

$$\{\}\ \mathsf{ht\_new}()\ \{ht.\ \exists \gamma_{\text{ht}}, \mathsf{IsHT}(ht, \gamma_{\text{ht}}) * \mathop{\Huge *}_{k:Key} m(\gamma, k, \mathsf{None})\}$$

$$\forall ht, \gamma_{\text{ht}}.\ \{\mathsf{IsHT}(ht, \gamma_{\text{ht}})\}\ \mathsf{ht\_free}(ht)\ \{\}$$

$$\forall ht, \gamma_{\text{ht}}, k, v.\ [\mathsf{IsHT}(ht, \gamma_{\text{ht}})]\ [m(\gamma_{\text{ht}}, k, v)]\ \{\}\ \mathsf{query}(ht, k)\ \{v'.\ v = v'\}$$

$$\forall ht, \gamma_{\text{ht}}, v, v'.\ [\mathsf{IsHT}(ht, \gamma_{\text{ht}})]\ \{m(\gamma_{\text{ht}}, k, v)\}\ \mathsf{update}(ht, k, v')\ \{m(\gamma_{\text{ht}}, k, v')\}$$

In order to define the IsHT and m propositions and prove the specifications, we once again start by defining a custom protocol of ghost state to represent the hash table's operation.

Step 1: Constructing the ghost resources. First, we create a custom ghost state that lets us relate the contents of a hash table's slots to the key-value pairs stored in the map, and which also encodes the appropriate kinds of state updates. Here, we just use "ordinary" PCM ghost state (§3.1), as we do not need to create any new $\twoheadrightarrow$ relations. Specifically, we can take the monoid $(\mathit{Key} \rightharpoonup \textsc{Excl}(\mathit{Value}^?)) \times (\mathbb{N} \rightharpoonup \textsc{Excl}((\mathit{Key} \times \mathit{Value})^?))$ and set,

$$m(\gamma, k, v) \triangleq \boxed{([k \mapsto \mathsf{ex}(v)], [])}^{\gamma} \qquad\qquad \mathit{slot}(\gamma, i, s) \triangleq \boxed{([], [i \mapsto \mathsf{ex}(s)])}^{\gamma}$$

For validity, $\mathcal{V}$, we encode invariants of the hash table: the key-value pairs and slot entries are consistent, the slots obey hash-probing invariants, and so on. (See the extended paper [Hance et al. 2023b] for full details.) We can then derive:

Linear-Probing Hash Table Resource

Propositions: $m(\gamma, k, v) \quad \mathit{slot}(\gamma, i, s) \quad$ (where $\gamma : \mathit{Name},\ k : \mathit{Key},\ v : \mathit{Value}^?,\ i : \mathbb{N},\ s : (\mathit{Key} \times \mathit{Value})^?$)

$$m(\gamma, k, v) * \mathit{slot}(\gamma, j, \mathsf{Some}((k, v_j))) \vdash v = \mathsf{Some}(v_j) \qquad (\textsc{QueryFound})$$

$$\frac{k \neq k_{H(k)}, \ldots, k_{i-1}}{m(\gamma, k, v) * \mathit{slot}(\gamma, i, \mathsf{None}) * (\mathop{\Large *}_{H(k) \le j < i} \mathit{slot}(\gamma, j, \mathsf{Some}(k_j, v_j))) \vdash v = \mathsf{None}} \ (\textsc{QueryNotFound})$$

$$m(\gamma, k, v) * \mathit{slot}(\gamma, j, \mathsf{Some}((k, v_j))) \Rrightarrow m(\gamma, k, v') * \mathit{slot}(\gamma, j, \mathsf{Some}((k, v'))) \qquad (\textsc{UpdateExisting})$$

$$\frac{k \neq k_{H(k)}, \ldots, k_{j-1}}{\begin{array}{c} m(\gamma, k, v) * \mathit{slot}(\gamma, i, \mathsf{None}) * (\mathop{\Large *}_{H(k) \le j < i} \mathit{slot}(\gamma, j, \mathsf{Some}(k_j, v_j))) \\ \Rrightarrow m(\gamma, k, v') * \mathit{slot}(\gamma, j, \mathsf{Some}((k, v'))) * (\mathop{\Large *}_{H(k) \le j < i} \mathit{slot}(\gamma, j, \mathsf{Some}(k_j, v_j))) \end{array}} \ (\textsc{UpdateInsert})$$

Now, what happens when we consider that the slot state $\mathit{slot}(\gamma, i, s)$ might be shared and read-only (via the reader-writer lock)? Fortunately, the query-related rules can easily be applied even if the *slot* or m state is shared. In particular, each query-related deduction concludes a pure proposition, so we can apply Unguard-Pers to obtain the predicate. The update-related rules, meanwhile, can just be applied normally using exclusively owned state (though we could, in principle, apply Guard-Upd to UpdateExisting if we needed to, since its $\mathop{\Large *}$ term remains unchanged by the update).

There is only one essential capability that the Hash Table Resource lacks as written. Specifically, we need to be able to *compose* different pieces of state from the Hash Table Resource. For example, if we have shared propositions $m(\gamma, k, v)$ and $\mathit{slot}(\gamma, j, \mathsf{Some}((k, v_j)))$, then to apply QueryFound, we actually need their *composition*, $m(\gamma, k, v) * \mathit{slot}(\gamma, j, \mathsf{Some}((k, v_j)))$. We can compose the shared state via Guard-And, but this only gives us a $\wedge$ conjunction. Therefore, we also need the following to complete the proofs for query and update:

Linear-Probing Hash Table Resource (Addendum)

$$m(\gamma, k, v) \wedge \mathit{slot}(\gamma, j, s) \ \vdash\ m(\gamma, k, v) * \mathit{slot}(\gamma, j, s)$$

$$\mathit{slot}(\gamma, b+1, s_{b+1}) \wedge (\mathop{\Large *}_{a \le j \le b} \mathit{slot}(\gamma, j, s_j)) \ \vdash\ (\mathop{\Large *}_{a \le j \le b+1} \mathit{slot}(\gamma, j, s_j)))$$

$$m(\gamma, k, v) \wedge (\mathop{\Large *}_{a \le j \le b} \mathit{slot}(\gamma, j, s_j)) \ \vdash\ m(\gamma, k, v) * (\mathop{\Large *}_{a \le j \le b} \mathit{slot}(\gamma, j, s_j))$$

Fortunately, in our PCM construction, these all follow from PCM-And: in particular, the hypothesis PCM-And is satisfied for elements x and y whenever the domains of the maps in x and y are disjoint.

Step 2: Verifying the implementation. Once again, we need to establish a definition for $\mathsf{IsHT}(\gamma_{\mathrm{ht}}, \mathit{ht})$, the shareable proposition that makes something into a hash table. Our hash table here is just made up of L reader-writer locks:

$$\mathsf{IsHT}(\gamma_{\mathrm{ht}}, \mathit{ht}) \triangleq \mathop{\Large *}_{0 \le i < L} \mathsf{IsRwLock}(\mathit{ht}.\mathit{locks}[i], \gamma_i, F_i)$$

```
$[\mathop{\Large\ast}_{0 \le j < L} \mathsf{IsRwLock}(ht.locks[j], \gamma_j, F_j)]\ [m(k, v)]$
$[\mathop{\Large\ast}_{H(k) \le j < i} slot(\gamma, j, \mathsf{Some}(k_j, v_j)) * (k \neq k_j)]$
    $\{\,\}$
    if $i \geq L$ then abort else
        lock_shared($ht.locks[i]$);
        $\{s.\ \mathsf{Sh}(\gamma_i, s)\}$
        $[(ht.slots[i] \hookrightarrow s) * slot(\gamma, i, s)]$
          let $r$ = (match $!ht.slots[i]$ with
                | None ⇒ None
                | Some(($k_i, v_i$)) ⇒
                    if $k = k_i$ then Some($v_i$)
                    else query_iter($ht, k, i + 1$)
          end) in
        $\{\mathsf{Sh}(\gamma_i, s) * (r = v)\}$
        unlock_shared($ht.locks[i]$);
        $\{r = v\}$
        $r$
```

Fig. 9. Proof outline of query_iter

where we let $F_i = \lambda s.\ (ht.slots[i] \hookrightarrow s) * slot(\gamma, i, s)$. We also roll up all the relevant ghost names into the super-name $\gamma_{\mathrm{ht}} = (\gamma, \gamma_0, \ldots)$ to serve as a name for the hash table as a whole.

The definition of F_i effectively says that the ith lock protects both the permission to access the ith slot of the hash table $ht.slots[i]$, but also the slot state in the ghost protocol.

Crucially, composing the IsRwLock propositions with $*$ makes IsHT easy to work with whether or not it is owned exclusively or shared. In particular, if we have $\mathsf{IsHT}(\gamma_{\mathrm{ht}}, ht)$ shared, as we expect of a concurrent hash table, then we can use Guard-Split to obtain a shared $\mathsf{IsRwLock}(ht.locks[i])$ so we can perform the usual lock operations (lock_shared, and so on).

For the proof, we focus on query here, as it makes the more interesting use of shared state. The meat of query is in the recursive query_iter. Our proof of query_iter is inductive, and it takes in its precondition the ghost state for all the slots previously accessed in the probe, in addition to the preconditions of query. Figure 9 shows a proof outline.

Stepping through, we first call lock_shared; using our shared $\mathsf{IsRwLock}(ht.locks[i], \gamma_i, F_i)$. From this call, we obtain $\mathsf{Sh}(\gamma_i, s)$, for some s which is fixed for the duration we hold the lock. By Rw-Shared-Guard and the definition of F_i, we have $\mathsf{Sh}(\gamma, s) \mathrel{\rightarrowtail\!\!\!\twoheadrightarrow}_{\gamma} (ht.slots[i] \hookrightarrow s) * slot(i, s)$ (eliminating the $\triangleright$ by Later-Guard). In the outline, we represent this shared state with our $[\ldots]$ notation.

Now, by Heap-Read-Shared we can perform the $!ht.slots[i]$ operation to load the value of s and case on it. If the slot is empty (None) then we apply QueryNotFound to get our answer; if the slot is full and the key matches, then we apply QueryFound. As discussed above, we have all we need to apply these deductions even when the state on the left-hand side is shared. The most interesting case is the recursive one: here, we append the newly obtained $slot(i, s)$ to obtain $\mathop{\Large\ast}_{H(k) \le j \le i} slot(\gamma, j, \mathsf{Some}(k_j, v_j))$, meeting the precondition for the recursive call.

The Client of the Hash Table. There are many options available to the hash table's client. We presume that the client wishes to share the hash table between threads, and she has the freedom to do this as she wishes. For instance, she might share it between a fixed number (N) of threads, using a fractional paradigm to give out a fraction $1/N$ to each (Example 3.2). Alternatively, she could allocate it permanently and share it forever (Example 3.4). Or she could put the hash table inside yet another reader-writer lock, with multiple threads able to concurrently access the hash table by taking a shared lock. This last possibility could be a step to augment our design with resizing: a client could take the lock exclusively to “stop the world” and rebuild the hash table.

Shared State with the Hash Table. One might wonder if the client could further apply Leaf and use the hash table to store propositions and manage shared access to them. For example, we might want to say $m(\gamma_{\mathrm{ht}}, k, v) \twoheadrightarrow\!\!\!\!\!\rightarrowtail F(k, v)$ for some proposition family F; then any client with shared access to the key k would also get shared access to the resource $F(k, v)$. Indeed, we can modify our Hash Table Resource to allow this. Specifically, we could reconstruct the resource via a storage protocol so we can prove $\twoheadrightarrow\!\!\!\!\!\rightarrowtail$ propositions. Effectively, the existing hash table monoid construction would become the protocol monoid for this new storage protocol.

6 MORE ADVANCED STORAGE PROTOCOLS

The lock example in our paper is intentionally kept somewhat simple for the sake of exposition. However, subsequent work has already used Leaf’s storage protocols to verify far more sophisticated read-sharing mechanisms. Specifically, IronSync [Hance et al. 2023c] is a verification framework that combines storage protocols with a handful of other techniques (notably, using a substructural type system to manipulate ghost resources, including shared ghost resources, rather than using CSL directly). Their framework embeds Leaf’s monoidal storage protocol definitions as axioms for manipulating their ghost resources.

They describe their experience using storage protocols to verify:

- A multi-counter reader-writer lock with additional, domain-specific features. This is a component of an effort to verify a multi-threaded page cache; the lock is used to protect a 4 KiB cache page, and the domain-specific features relate to reading and writing the cache page from disk. The lock not only allows read-sharing of memory resources for the 4 KiB pages, but also of ghost resources related to their contents.
- A concurrent ring buffer with multiple producers and multiple consumers, where entries are alternately writeable and read-shared, as producer threads enqueue messages to be read (possibly simultaneously) by a number of consumer threads. This is a component of a state replication algorithm [Calciu et al. 2017], targeting non-uniform memory access (NUMA) architecture. Once again, this not only allows read-sharing of memory resources, but also ghost resources related to the operation log.

The second example, in particular, demonstrates that read-sharing protocols extend beyond reader-writer locks. Furthermore, both examples (similar to our hash table) demonstrate the use of read-shared custom ghost resources.

7 SOUNDNESS

Here, we sketch our construction of the Leaf logic within the Iris separation logic; for full details, consult the Coq development. To get the most out of this section, it helps for the reader to be already familiar with Iris; Jung et al. [2018] provide all necessary background.

As context, we review the components of Iris. First, there is the *Iris base logic*, a step-indexed logic of resources in the abstract, with no primitive notion of a program or Hoare logic. The Iris base logic is proved sound via a semantic model called *the UPred model*. Then, atop the base logic, Iris can do a variety of useful things, e.g., instantiate a program logic given some operational semantics.

To build Leaf, we add a few minor deduction rules to the base logic, proved sound via the *UPred* model. Our additions to the base logic are given in blue. We then define $\twoheadrightarrow\!\!\!\!\!\rightarrowtail$, $\langle p \rangle^{\gamma}$, and $\mathrm{sto}(\gamma, F)$, and prove all of Leaf’s deduction rules within the Iris logic. The rest of the Iris framework, such as the machinery to instantiate a program logic and prove adequacy theorems, is unchanged.

Ghost state. In Iris, ghost state is constructed from a mathematical object called a *CMRA*. A PCM is a special case of a discrete CMRA, and the ghost state ($\boxed{x}^{\gamma}$) in this paper is just the usual Iris ghost state. We also add the PCM-And rule to the base logic, which follows straightforwardly from the definition of $\wedge$ over the *UPred* model and holds for any discrete CMRA.

Invariants. Our definitions build on Iris's invariants, so we review those here. Iris defines (within the base logic) a persistent proposition $\boxed{P}^{\iota}$ as knowledge that an invariant P is allocated at name ι. Iris then proves the following rules so the user can allocate, open, and close invariants:

$$(\mathcal{N} \text{ infinite}) \vdash \triangleright P \Rrightarrow_{\mathcal{E}} \exists \iota.\ (\iota \in \mathcal{N}) * \boxed{P}^{\iota} \qquad \text{(Inv-Alloc)}$$

$$(\iota \notin \mathcal{E}) * \boxed{P}^{\iota} \vdash \mathsf{True}\ {}^{\mathcal{E}\cup\{\iota\}}\!\Rrightarrow^{\mathcal{E}} (\triangleright P) \qquad \text{(Inv-Open)}$$

$$(\iota \notin \mathcal{E}) * \boxed{P}^{\iota} \vdash (\triangleright P)\ {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\cup\{\iota\}} \mathsf{True} \qquad \text{(Inv-Close)}$$

The definition of $⤔$. To define $⤔$, we first define a "pre-guards" operator, $\hat{⤔}$, for *finite* sets ζ, and then take the closure under supersets to define the real $⤔$.

$$\begin{aligned}
\mathsf{AllocatedInvs}(\zeta) &\triangleq \mathop{\Large *}_{\iota\in\zeta} \exists P : \mathit{iProp}.\ \boxed{P}^{\iota} \\
\mathsf{OwnInvs}(\zeta) &\triangleq \mathop{\Large *}_{\iota\in\zeta} \exists P : \mathit{iProp}.\ \boxed{P}^{\iota} * P \\
P \mathrel{\hat{⤔}_{\zeta}} Q &\triangleq \mathsf{AllocatedInvs}(\zeta) * \\
&\qquad\quad \forall R : \mathit{iProp}.\ \square(P * (P \mathrel{-\!\!*} \mathsf{OwnInvs}(\zeta) * R) \mathrel{-\!\!*} \diamond (Q * (Q \mathrel{-\!\!*} \mathsf{OwnInvs}(\zeta) * R))) \\
P \mathrel{⤔_{\mathcal{E}}} Q &\triangleq \exists \zeta.\ (\zeta \subseteq \mathcal{E}) * (P \mathrel{\hat{⤔}_{\zeta}} Q)
\end{aligned}$$

The $\hat{⤔}$ definition can be read roughly as, *if we have the invariants at* ζ *and some other state* R, *which can separate into* P *and some other component, then it also can separate into* Q. Note that this definition does not perform an update ($|\!\dot{\Rrightarrow}$) or even a mask change. If we used $|\!\dot{\Rrightarrow}$ we would not be able to show Guard-And because it would require us to perform two potentially contradictory updates simultaneously. It is also important that the definition preserves $\mathsf{OwnInvs}(\zeta)$, so that we can prove Guard-Trans without needing an additional disjointness condition. Finally, taking the closure under supersets lets us prove Guard-Weaken-Mask.

Basic $⤔$ *proofs.* From this definition of $⤔$, many Leaf rules are straightforward: Guard-Refl, Guard-Trans, Guard-Split, Guard-Weaken-Mask, Guard-Pers, and Unguard-Pers. To prove Guard-Upd, we perform a view shift from $\mathcal{E}_1 \cup \mathcal{E}_2$ to $\mathcal{E}_1$, opening all the invariants (Inv-Open) in some $\zeta \subseteq \mathcal{E}_2$. This is primarily where we use the finiteness of ζ and the $\mathsf{AllocatedInvs}(\zeta)$ term. Later-Guard follows thanks to our use of $\diamond$ in the definition of $⤔$. Later-Pers-Guard then follows by pulling out the persistent part with Unguard-Pers, and then applying Later-Guard.

Point Propositions. Finally, we come to Guard-Implies and Guard-And. The inherent difficulty here is that $A \vdash P$ does not in general imply A can separate into $P * (P \mathrel{-\!\!*} A)$. We therefore need to use the fact that P is a point proposition. The key is that point propositions are all of the form $\mathsf{Own}\ (t)$ where t is an element of the *global* CMRA instantiating the Iris model. We can show:

$$A * (A \mathrel{-\!\!*} \mathsf{Own}\ (t)) \vdash \mathsf{Own}\ (t) * (\mathsf{Own}\ (t) \mathrel{-\!\!*} A) \qquad \text{(Split-Own)}$$

This rule is somewhat peculiar: initially it looks like it must be wrong, since applying A and $A \mathrel{-\!\!*} P$ ought to give you just P, not P plus something else. On second glance, it makes some intuitive sense that you ought to be able to "go back" to A, since you had A to begin with. At any rate, it does at least hold for the $\mathsf{Own}\ (t)$ propositions, and it follows from the model definitions of $*$, $\mathrel{-\!\!*}$, and $\mathsf{Own}\ (t)$. Reynolds [2008] proved a similar theorem about what they called *strictly exact assertions*; strictly exact assertions are not representable in Iris, but $\mathsf{Own}\ (t)$ propositions play a similar role.

To finish the proofs, we need something to account for our use of $\diamond$ in the definition of $⤖$.

$$A * (A \mathrel{-\!\!*} \diamond\,\mathsf{Own}\,(t)) \vdash (\diamond\,\mathsf{Own}\,(t)) * (\mathsf{Own}\,(t) \mathrel{-\!\!*} A) \qquad \text{(Split-Own-Except0)}$$

From this, the proofs of Guard-Implies and Guard-And follow.

Storage Protocols. Given an arbitrary storage protocol (Figure 5a), we can define a discrete CMRA $\textsc{Prot}(P) \triangleq \mathrm{prot}(P) \mid \epsilon$ where ϵ is the unit and $\mathrm{prot}(x) \cdot \mathrm{prot}(y) = \mathrm{prot}(x \cdot y)$. We define validity on this algebra in terms of C, specifically, $\mathcal{V}(\epsilon) = \mathrm{True}$ and $\mathcal{V}(\mathrm{prot}(x)) = \exists y.\, C(x \cdot y)$. Then we construct ghost state via the authoritative-fragmentary construction, $\textsc{Auth}(\textsc{Prot}(P))$. Let:

$$\mathrm{sto}(\gamma, F) \triangleq \mathrm{RespectsComposition}(F) * \boxed{\exists (x : P).\, \boxed{\bullet\ \mathrm{prot}(x)}^{\gamma} * C(x) * F(\mathcal{S}(x))}^{\gamma} * \boxed{\circ\ \mathrm{prot}(\epsilon)}^{\gamma}$$
$$\langle p \rangle^{\gamma} \triangleq \boxed{\circ\ \mathrm{prot}(p)}^{\gamma}$$

In other words, the sto predicate gives us the invariant at location γ, which contains the authoritative copy of some initialized protocol along with any state currently stored in the protocol.

We can now proceed with the proofs of the laws in Figure 5b. To prove SP-Exchange, we use our knowledge of the invariant from $\mathrm{sto}(\gamma, F)$ to open it, obtaining the stored state $F(\mathcal{S}(x))$ and the authoritative knowledge of the protocol state, $\boxed{\bullet\ \mathrm{prot}(x)}^{\gamma}$. We perform the update and then close the invariant. SP-Deposit, SP-Withdraw, and SP-Update are just special cases of SP-Exchange.

For SP-Guard, we need to prove a $⤖$ proposition. Starting with the LHS of the $\hat{⤖}$ definition, we have $\langle p \rangle^{\gamma} * (\langle p \rangle^{\gamma} \mathrel{-\!\!*} \mathsf{OwnInvs}(\zeta) * R)$. This entails $\boxed{\bullet\ \mathrm{prot}(x)}^{\gamma} \wedge \boxed{\circ\ \mathrm{prot}(p)}^{\gamma}$ for some x. Using PCM-And, we then obtain that $p \preceq x$, apply the hypothesis that $p ⤀ s$, and pull $F(s)$ out of $F(\mathcal{S}(x))$.

Mechanization. The above definitions and proofs are formalized in Coq, on top of the Iris library, available in our artifact [Hance et al. 2023a]. Our Coq formalization also includes the instantiation of Leaf on a heap-based language with atomic reads and writes (§3.3), and the proof of the lock-based hash table (§4 and §5).

8 RELATED WORK AND COMPARISONS

Shared, read-only ownership in separation logic. Fractional permissions and counting permissions are existing mechanisms for temporarily shared read-only state that have appeared in a variety of contexts. It is well-established that one can represent these via monoidal ghost state, but to our knowledge, the existing approaches do not provide a uniform way to reason about their interpretations as read-only state the way Leaf's $⤖$ operator does. Below, we examine in depth what our case study would look like if we used traditional fractional resources.

Fractions have also been used in Iris's *cancellable invariants*, i.e., invariants which are temporarily shared and then reclaimed, just as Leaf's guarded propositions are. However, Iris's cancellable invariants do not support overlapping conjunction the way Leaf's $⤖$ does.

Fractions have also been used for a variety of more sophisticated applications, such as RustBelt's lifetime logic [Jung et al. 2017] which can reason about Rust's shared borrows. Again using fractions, Dang et al. [2020] show how to handle shared resource reclamation in relaxed memory settings. We leave it as future work to see if Leaf's more general protocols can be adapted to those domains.

Charguéraud and Pottier [2017] introduce a "temporary read-only modality" for the sequential setting, allowing a user to temporarily exchange a resource for a read-only resource, managed by a lexical scope rule. In Leaf, shared resources are not necessarily bound to a fixed scope: guard propositions can be conditional and their scope dynamically determined.

Some prior work has also demonstrated more general permission logics capturing fractional and counting use cases [Parkinson 2005], some with arbitrary composition structure [Dockins et al. 2009]. However, these methods require *all* state from the permission logic to be collected in order to regain exclusive access. To our knowledge, they are not flexible enough to support protocols like our RwLock protocol, which must represent nontrivial states even when no proposition is stored. Meanwhile, fictional separation logic [Jensen and Birkedal 2012] uses a general extension mechanism similar to Leaf's relationship between protocol and storage monoids, though it does not address general read-sharing mechanisms.

Concurrent separation logics with custom ghost state. Many CSL frameworks have introduced custom ghost state, i.e., mechanisms for users to define new resources, such as the work on Concurrent Abstract Predicates (CAP) [Dinsdale-Young et al. 2010; Svendsen and Birkedal 2014], Fine-grained Concurrent Separation Logic (FCSL) [Nanevski et al. 2014] with its concurroids (also based on PCMs), and Iris with its CMRAs (see below). To our knowledge, none of these frameworks have yet been used to provide a modular representation of temporarily-shared custom resources that supports the composition of resources shared via the representation.

Iris's ghost state formalism. Leaf creates a custom ghost state mechanism, storage protocols, based on PCM ghost state. Iris has generalized PCM ghost state in a different direction, creating an algebraic object called a *CMRA* [Jung et al. 2016], which has two new features over PCMs. The first is a built-in notion of *persistent* state; in Leaf, we de-emphasize persistent state because of our focus on temporarily-shared state. However, it would be straightforward to incorporate persistence into the protocol monoid formalism. The second is a *step-indexed* notion of equality, which makes CMRAs suitable for *higher-order* ghost state and many of the foundational elements of Iris. In Leaf, we wanted our storage protocols to be easily represented in first-order logic with a discrete notion of equality. Factoring our map into two steps, $\mathcal{S} : P \rightarrow S$ and $F : S \rightarrow iProp$ is what allows us to define a storage protocol without any step-indexing. As such, one can understand storage protocols as a particular ghost state abstraction built on CMRA machinery.

Verified hash tables. Hash tables have been verified before [Ho and Protzenko 2022; Polikarpova et al. 2017; Pottier 2017], including a concurrent one done in Iris that uses mutual exclusion locks [Clausen 2017]. Our concurrent hash table has some crucial differences that make it interesting: **(i)** we use reader-writer locks, and thus shared ownership for queries, and **(ii)** ours requires a single operation (update or query) to take more than one lock.

Case study comparison. It is worth comparing explicitly to how our reader-writer lock and hash table case study might be done if we used more traditional techniques. One of the most common such techniques in use to day is—as we have referenced several times in this paper—the technique of fractional permissions. If we were to build the hash table case study using fractional permissions (in Iris, or in any other framework supporting monoid ghost state and invariants), it might look something like the following:

- First, the resource being protected by the lock would need to have a built-in notion of being fractional. The reader-writer lock spec could be parameterized over a fractional proposition family, $F(x, q)$. The $\mathsf{IsRwLock}(rw, \gamma, F)$ proposition would need to be fractionalized as well, which could be done using a technique called *cancellable invariants* (invariants with associated fractional tokens, which allow the inner resources to be reclaimed). Ultimately, the lock's Hoare triples would look something like (to select a few):

$$\forall rw, \gamma, F, q_0.\ \{\mathsf{IsRwLock}(rw, \gamma, F, q_0)\}\ \mathsf{lock\_shared}(rw)\ \{\mathsf{IsRwLock}(rw, \gamma, F, q_0) * \exists x, q.\ \mathsf{Sh}(\gamma, x, q) * F(x, q)\}$$
$$\forall rw, \gamma, F, x, q_0, q.\ \{\mathsf{IsRwLock}(rw, \gamma, F, q_0) * \mathsf{Sh}(\gamma, x, q) * F(x, q)\}\ \mathsf{unlock\_shared}(rw)\ \{\mathsf{IsRwLock}(rw, \gamma, F, q_0)\}$$
$$\forall rw, \gamma, F.\ \{\mathsf{IsRwLock}(rw, \gamma, F, 1)\}\ \mathsf{rwlock\_free}(rw)\ \{\}$$

Furthermore, the lock would need to guarantee that,

$$\mathsf{IsRwLock}(rw, \gamma, F, q_0) * \mathsf{IsRwLock}(rw, \gamma, F, q_1) \dashv\vdash \mathsf{IsRwLock}(rw, \gamma, F, q_0 + q_1)$$

while the client would have to promise that $F(x, q_0) * F(x, q_1) \dashv\vdash F(x, q_0 + q_1)$. Also observe that Sh to track the fractional amounts that are "lent out," so we can make sure the same amount is returned later.

- To verify the reader-writer lock, we would internally define an invariant that maintains some possibly-fractional amount of the resource, so that it has something to "lend out" whenever a client takes a read lock. Further, we would need to create ghost resources to define $\mathsf{Sh}(\gamma, x, q)$, $\mathsf{Exc}(\gamma)$, intermediate states, and so on. These resources would need to keep track of all the fractional amounts that are "lent out," and make sure they sum to the correct amount. We would still need to reason about the intermediate states of the locking operations, but now the relationships are slightly harder to specify, because they interact with all of the fractional accounting.
- The client of the lock (the hash table) would need to make sure the resources it uses have a built-in fractional notion so they can interoperate with the lock. Thus the points-to operations would need a built-in notion of fractions ($\ell \xhookrightarrow{\text{frac}}_q v$) while the hash table's "slot resources" $\mathit{slot}(\gamma, i, s)$ would be replaced by fractional resources $\mathit{slot}(\gamma, i, s, q)$. Now, all the updates and deductions would be expressed with fractions, and proving the $\rightsquigarrow$ relations would involve reasoning about a composition operator $\cdot$ that adds rational numbers. For example, one has to reason like, "Suppose I have a q fraction of slot j, and a unit amount of slot $j + 1$, and I replace slot $j + 1$ with a unit amount of a new slot value..."

While this is all certainly possible, our perspective is that it involves a large number of "bureaucratic" details that do not directly relate to the programmer's primary intuition of why the program is correct. By contrast, when doing this in the Leaf style, as we have seen:

- The RwLock specification—that is, the "interface" between the two components—becomes cleaner. Neither $\mathsf{IsRwLock}$, Sh, nor F need an additional rational number parameter (Figure 1). Instead, the relationships between these components and the sharing that takes place are all made clear through the ⤖ and $[\ldots]\ \{\ldots\}\ e\ \{\ldots\}$ notation.
- The RwLock is easier to verify because the storage protocol formulation helps us reduce the problem to a series of proof obligations regarding the evolution of the system.
- The Hash Table is easier to verify in Leaf because we can reason in a manner similar to how we would do it in an exclusive ownership setting, without the encoding having to "bake in" sharing-related details. For example, we would reason something like, "Suppose I have slot j and slot $j + 1$, and I replace slot $j + 1$..." and then rely on Leaf to apply this in the presence of shared slots. This sort of simplification would apply to any application that uses fine-grained reader-writer locks in a similar manner.

9 CONCLUSION

We have introduced Leaf, a concurrent separation logic with an approach to temporarily shared ownership based on our novel guarding operator, ⤖. We showed that Leaf can help the user implement and verify sharing strategies, that it allows modular specifications involving shared state that abstract away the sharing mechanism being used, and that Leaf's composition capabilities allow it to handle fine-grained concurrency.

ACKNOWLEDGMENTS

We thank Tej Chajed and the anonymous reviewers for helpful feedback.

Work at CMU was supported, in part, by an Amazon Research Award (Fall 2022 CFP), a gift from VMware, the Future Enterprise Security initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab), and the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award No. CNS-1700521.

REFERENCES

Aleš Bizjak and Lars Birkedal. 2018. On Models of Higher-Order Separation Logic. *Electronic Notes in Theoretical Computer Science* 336 (2018), 57–78. https://doi.org/10.1016/j.entcs.2018.03.016 The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).

Richard Bornat, Peter O'Hearn, and Matthew Parkinson. 2005. Permission Accounting In Separation Logic. *Sigplan Notices - SIGPLAN* 40, 259–270. https://doi.org/10.1145/1047659.1040327

John Boyland. 2003. Checking Interference with Fractional Permissions. *SAS*, 55–72. https://doi.org/10.1007/3-540-44898-5_4

Irina Calciu, David Dice, Yossi Lev, Victor Luchangco, Virendra Marathe, and Nir Shavit. 2013. NUMA-Aware Reader-Writer Locks. *ACM SIGPLAN Notices* 48. https://doi.org/10.1145/2442516.2442532

Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 15 pages. https://doi.org/10.1145/3037697.3037721

Arthur Charguéraud and François Pottier. 2017. Temporary Read-Only Permissions for Separation Logic. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 260–286. https://doi.org/10.1007/978-3-662-54434-1_10

E. Clausen. 2017. Verifying Hash Tables in Iris.

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt Meets Relaxed Memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

Dave Dice and Alex Kogan. 2019. BRAVO: Biased Locking for Reader-Writer Locks. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 315–328.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 287–300. https://doi.org/10.1145/2429069.2429104

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A Fresh Look at Separation Algebras and Share Accounting. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*. https://doi.org/10.1007/978-3-642-10672-9_13

Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis. 2019. Lock–Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems. *ACM Trans. Comput. Syst.* 36, 1, Article 1 (March 2019), 149 pages. https://doi.org/10.1145/3301501

Travis Hance, Jon Howell, Oded Padon, and Bryan Parno. 2023a. *Leaf: Modularity for Temporary Sharing in Separation Logic (Artifact)*. https://doi.org/10.5281/zenodo.8327489

Travis Hance, Jon Howell, Oded Padon, and Bryan Parno. 2023b. Leaf: Modularity for Temporary Sharing in Separation Logic (Extended Version). (2023). https://doi.org/10.48550/arXiv.2309.04851

Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. 2023c. Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 911–929. https://www.usenix.org/conference/osdi23/presentation/hance

Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (aug 2022), 31 pages. https://doi.org/10.1145/3547647

W.C. Hsieh and W.E. Weihl. 1992. Scalable Reader-Writer Locks For Parallel Systems. In *Proceedings Sixth International Parallel Processing Symposium*. 656–659. https://doi.org/10.1109/IPPS.1992.222989

Jonas Braband Jensen and Lars Birkedal. 2012. Fictional Separation Logic. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 377–396. https://doi.org/10.1007/978-3-642-28869-2_19

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2 (12 2017), 1–34. https:

//doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, R. Krebbers, Jacques-Henri Jourdan, A. Bizjak, L. Birkedal, and Derek Dreyer. 2018. Iris From The Ground Up: A Modular Foundation For Higher-Order Concurrent Separation Logic. *Journal of Functional Programming* 28 (2018). https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-Aware Blocking Synchronization Primitives. In *Proceedings of the USENIX Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 603–615.

Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., USA.

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*. Springer-Verlag, Berlin, Heidelberg, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Neelakantan Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. 2012. Superficially Substructural Types. *ACM SIGPLAN Notices* 47, 41–54. https://doi.org/10.1145/2364527.2364536

Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective Auxiliary State for Coarse-Grained Concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 561–574. https://doi.org/10.1145/2429069.2429134

Ran Liu, Heng Zhang, and Haibo Chen. 2014. Scalable Read-Mostly Synchronization Using Passive Reader-Writer Locks. In *Proceedings of the USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 219–230.

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

Peter W. O'Hearn. 2007. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.* 375, 1–3 (April 2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

Matthew Parkinson. 2005. *Local Reasoning for Java.* Ph. D. Dissertation. University of Cambridge.

Nadia Polikarpova, Julian Tschannen, and Carlo Furia. 2017. A Fully Verified Container Library. *Formal Aspects of Computing* 30 (09 2017), 1–29. https://doi.org/10.1007/s00165-017-0435-1

François Pottier. 2017. Verifying a Hash Table and Its Iterators in Higher-Order Separation Logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (Paris, France) *(CPP 2017)*. Association for Computing Machinery, New York, NY, USA, 3–16. https://doi.org/10.1145/3018610.3018624

John Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

John C. Reynolds. 2008. An Introduction to Separation Logic. https://www.cs.cmu.edu/~jcr/copenhagen08.pdf Accessed: 2022-11-08.

Jun Shirako, Nick Vrvilo, Eric G. Mercer, and Vivek Sarkar. 2012. Design, Verification and Applications of a New Read-Write Lock Algorithm. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Pittsburgh, Pennsylvania, USA) *(SPAA '12)*. Association for Computing Machinery, New York, NY, USA, 48–57. https://doi.org/10.1145/2312005.2312015

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

The cppreference Team. 2011. std::lock_guard. https://en.cppreference.com/w/cpp/thread/lock_guard Accessed: 2022-11-08.

The Rust Team. 2014. Struct std::sync::RwLockReadGuard. https://doc.rust-lang.org/std/sync/struct.RwLockReadGuard.html Accessed: 2022-11-08.

Received 2023-04-14; accepted 2023-08-27