In addition to pre- and postconditions, program specifications in recent separation logics for concurrency have employed an algebraic structure of *resources*—a form of state transition systems—to describe the state-based program invariants that must be preserved, and to record the permissible atomic changes to program state. In this paper we introduce a novel notion of *resource morphism*, i.e. structure-preserving function on resources, and show how to effectively integrate it into separation logic, using an associated notion of morphism-specific *simulation*. We apply morphisms and simulations to programs verified under one resource, to compositionally adapt them to operate under another resource, thus facilitating proof reuse.

CCS Concepts: • Theory of computation → Separation logic; Hoare logic; Type theory; • Software and its engineering → Formal software verification; • Computing methodologies → Concurrent algorithms.

Additional Key Words and Phrases: Program Logics for Concurrency, Hoare/Separation Logics, Coq

1 INTRODUCTION

The main problem when formally reasoning about concurrent data structures is achieving *compositionality* of proofs: how to ensure that methods of a data structure, once verified, can be used in a larger context without re-verification. There exist many solutions to the problem, roughly divided into two kinds: *linearizability* [Herlihy and Wing 1990], or more generally *contextual refinement* [Filipović et al. 2010a; Liang and Feng 2018; Liang et al. 2014], and *Concurrent Separation Logic (CSL)* [Brookes 2007; O’Hearn 2007], and its many recent extensions to fine-grained (i.e., lock-free) concurrency [da Rocha Pinto et al. 2014; Dinsdale-Young et al. 2010; Jung et al. 2015; Nanevski et al. 2014; Svendsen and Birkedal 2014; Svendsen et al. 2013]. More recently, some approaches [Frumin et al. 2018; Turon et al. 2013] employed variants of separation logic to establish linearizability and contextual refinement themselves, suggesting separation logic as a general-purpose method for reasoning about concurrent programs.

On the other hand, composition is also the cornerstone of *type theory*, where types serve as the interface that abstracts the internal properties of programs and proofs. Because both are focused on...
composition, type theory and separation logic are closely related. For example, from the inception of (sequential) separation logic, it has been understood [O’Hearn et al. 2001; Reynolds 2002] that its reasoning power arises from the key property of fault avoidance, which implies the features usually associated with separation logic, such as framing and small-footprint semantics. Fault avoidance states that a program verified against some pre- and postcondition, doesn’t crash (say, by reading from a deallocated pointer), if started in a state satisfying the precondition. This has inspired a stateful type theory [Nanevski 2016; Nanevski et al. 2006], where the type ascription

\[ e : \{P\}\{Q\} \]

denotes that the program \( e \) has a precondition \( P \) and a postcondition \( Q \) (both predicates over program states), in the sense of partial correctness. The Hoare type \( \{P\}\{Q\} \) is a form of dependently-typed state monad, indexed by \( P \) and \( Q \), which encapsulates the effects of state and divergence, similarly to monads in Haskell. In the typed setting, fault avoidance is forced onto the formalism by the requirement that “well-typed programs cannot go wrong” [Milner 1978]. Thus, Hoare types give rise to not just a Hoare logic, but separation logic specifically. In other words, separation logic is a type theory of state. Hoare types also enable a formulation and low-overhead implementation [Nanevski et al. 2008; Svendsen et al. 2011] of separation logic as an extension of, or a shallow embedding into, a standard type theory (e.g. Coq).

Using the above connection as a guiding principle, this paper derives a type-based formulation of separation logic for fine-grained concurrency. Immediately motivated by the form of typeful specification for fine-grained programs, our contributions are two novel and foundational abstractions for compositional verification—the morphisms and the simulations from the paper’s title—and a way to incorporate them into Hoare-style reasoning by means of a single inference rule. The upshot is conceptually simple foundations for separation logic for fine-grained concurrency.

1.1 Resources

As proposed by Dinsdale-Young et al. [2010], and utilized in different ways in many recent formalisms [da Rocha Pinto et al. 2014; Jung et al. 2015; Nanevski et al. 2014; Svendsen and Birkedal 2014; Svendsen et al. 2013], the key technical requirement that fine-grained concurrency imposes on a Hoare-style logic is enriching the Hoare specifications with state transition systems (STS) of a specific form—termed resources [Hoare 1972; O’Hearn 2007; Owicki and Gries 1976] in this paper. For example, in our type-based setting, we extend the Hoare type with a resource \( V \), as in

\[ e : \{P\}\{Q\} @ V \]

to signify that \( e \) has a precondition \( P \) and a postcondition \( Q \), but also that the atomic state changes that \( e \) may carry out are circumscribed by the transitions of \( V \). We also say that \( e \) is typed by \( V \), that \( e \) inhabits \( V \), or that \( e \) is in \( V \).

Two programs can be composed in parallel (or sequentially), only if they are typed by the same resource. Thus, the resource in the type annotation bounds the interference that concurrent threads can perform on each other’s execution, which is essential for reasoning about the composition.\(^1\)

To quickly illustrate resources in our particular setting, consider a spin lock \( r \) (a shared Boolean pointer) and a program that locks \( r \) by setting it to true, and loops if \( r \) is already set: \(^2\)

\[
\text{lock } \equiv \text{ do } x \leftarrow \text{CAS}(r, \text{false}, \text{true}) \text{ while } \neg x
\]

\(^1\)The idea of bounding the interference is the foundation behind the classic rely-guarantee method [Jones 1983] as well. In fact, resources may be seen as structuring and compactly representing—in the form of transitions—the rely and guarantee relations of the rely-guarantee method.

\(^2\)The Compare-and-Set variant of CAS(\( r, a, b \)) [Herlihy and Shavit 2008] atomically sets the pointer \( r \) to \( b \) if \( r \) contains \( a \), otherwise leaves \( r \) unchanged. It moreover returns a Boolean value denoting the success or failure of the operation.
Figure 1 shows (an abstracted form of) the resource Spin, suitable to type lock. Every execution of lock describes a path through Spin consisting of several idle transitions id_tr, corresponding to unsuccessful CAS’s, followed by a locking transition lock_tr corresponding to a successful CAS. Similarly, the Spin resource also types the unlock program

\[ \text{unlock} \equiv r := \text{false} \]

which may be seen as taking the unlock_tr transition if \( r \) stores true, or staying idle if \( r \) stores false to begin with. Because lock and unlock are typed by the same resource, they can be composed, sequentially or in parallel. The typing guarantees that the concurrent environment of lock and unlock is bound to only ever execute the transitions of Spin, and can’t cause "surprises", such as deallocating the pointer \( r \) while lock or unlock are executing.

### 1.2 Morphisms

This brings us to the first technical contribution of the paper. As soon as resources are introduced into types, it becomes necessary to coerce a program from one type (i.e., resource) to another.

As one example of coercion, consider two procedures, inhabiting different resources, each specifying its own concurrent data structure (say, a stack and a queue). If we want to use the stack and the queue together in a program, we must coerce the procedures into a common resource that includes the functionality of both structures, and describes how the two interact.

As another example, consider refining the behavior of already implemented resources. Suppose we want to use the bare-bones spin locks described by Spin to develop more sophisticated locking protocols: CSL-style mutually exclusive locks [O’Hearn 2007], or non-mutually-exclusive locks such as readers-writers locks [Bornat et al. 2005; Courtois et al. 1971] where a reader acquires a lock to allow access to multiple readers, but not writers. Both developments can be seen as extending Spin with additional ghost state to represent the invariants of the refined locking protocol, and then coercing the lock and unlock procedures to modify this additional ghost state at the precise moment when lock transitions by lock_tr, and unlock transitions by unlock_tr. The developments thus compositionally reuse the definitions and proofs of lock and unlock, each in its own way. We carry out the first development in Section 4, and the second in the Coq code [Nanevski et al. 2019a].

To achieve coercion we introduce morphisms between resources. A resource morphism \( f : V \rightarrow W \) is a structure-preserving mapping from resource \( V \) to resource \( W \), which acts on a program \( e \) typed by \( V \), to derive a program typed by \( W \), essentially by re-interpreting the \( V \)-transitions that \( e \) takes, as \( W \)-transitions.

Morphisms arise naturally, because a structure in mathematics typically is associated with an appropriate notion of a structure-preserving function. Examples abound: vector spaces and linear maps, groups and their homomorphisms, complete partial orders and continuous functions, functors.

---

3 We'll define the state space and the transitions of Spin in Section 2, and eventually tie them to the implementations of lock and unlock. At this point, it suffices to know that a program in Spin may transition by lock_tr only if \( r \) is free (thereby locking it), and by unlock_tr only if \( r \) is locked (thereby freeing it).
and natural transformations, etc. Morphisms endow the structure with dynamics and allow studying it under change. This will be the case for us as well.

More specifically, and akin to how automata homomorphisms [Ginzburg 1968] are defined componentwise, a resource morphism \( f : V \rightarrow W \) consists of two partial functions \( f_\Sigma \) and \( f_\Delta \), acting respectively on states (\( \Sigma \)) and transitions (\( \Delta \)). \( f_\Sigma \) takes a state in \( W \) and produces a state, if defined, in \( V \) (note the contravariance); and \( f_\Delta \) takes a state in \( W \) and transition in \( V \) and produces a transition, if defined, in \( W \). Combined, \( f_\Sigma \) and \( f_\Delta \) act on a program \( e \) inhabiting \( V \) to produce a program inhabiting \( W \), using the following process.

Referring to Figure 2, the morphed program inhabits \( W \), so we describe it starting with a \( W \)-state \( s_w \) in the lower-right corner of the diagram (the predicate \( I \) that is applied to \( s_w \) in the diagram will be explained promptly). To compute the next state of the morphed program, we first take \( s_v = f_\Sigma s_w \) which is a state in \( V \) (utilizing contravariance of \( f_\Sigma \)). If \( e \) takes a transition \( t_v \) to step from \( s_v \) to \( s'_v \) in \( V \), the corresponding transition of the morphed program is \( t_w = f_\Delta s_w t_v \). If \( t_w \) steps from \( s_w \) to \( s'_w \), the process is repeated for \( s'_w \) and the next transition of \( e \).

The morphing process determines a program in \( W \) that we denote morph \( f e \). Here, morph is a program constructor, and a form of function application of \( f \) to \( e \). In the sequel, we introduce the infrastructure to program (and prove!) with morphisms and morphed programs.

### 1.3 Simulations and Inference

To reason about morph \( f e \), we introduce our second contribution: morphism-specific simulations. An \( f \)-simulation (or simply, a simulation, when \( f \) is clear from the context) is a predicate \( I \) over \( W \)-states that acts like a loop invariant for the iterative process of morphing by \( f \). Specifically, an \( f \)-simulation satisfies, among other conditions presented in Section 3, the key technical property that the diagram in Figure 2 commutes. Given \( s_w, s_v, t_v \) and \( s'_v \) that partially describe the diagram, if \( I s_w \), then there exists a state \( s'_w \) that completes the diagram: \( t_w = f_\Delta s_w t_v \) exists, \( t_w s_w s'_w, I s'_w, \) and \( f_\Sigma s'_w = s'_v \). Because \( I s'_w \) holds, simulation \( I \) is preserved by \( f \).

Simulations provide a way to reason about morph \( f e \) compositionally, i.e., out of \( e \)'s type, achieving our ultimate goal of program and proof re-usability. The specifics are prescribed by the following single inference rule, which is our third contribution:

\[
\text{morph } f e : \{P\} \{Q\}@V \\
\text{Morph } \text{morph } f e : \{\lambda s_w. f^\gamma P s_w \land I s_w\} \{\lambda s_w. f^\gamma Q s_w \land I s_w\}@W \\
\text{where } f^\gamma R s_w \equiv \exists s_v. s_v = f_\Sigma s_w \land R s_v
\]

The Morph rule translates the requirements for a morphism \( f : V \rightarrow W \), and a simulation \( I \), expressed diagrammatically in Figure 2, from properties of states and transitions of the resources \( V \) and \( W \), into the specification of a program morph \( f e \) in \( W \). The latter spec is structured, both in its pre- and the postconditions, as a conjunction of (i) the transformation of \( e \)'s spec from \( V \) to \( W \),
and (ii) a statement that the simulation $I$, being in fact an invariant of the resources’ transitions, holds at the boundaries of the morphed execution of $e$. For the first part, we use the predicate $f \cdot \_\_\_\_$ to lift the pre- and postconditions of $e$ from states in $V$ to states of $W$, which are related via $f_{\Sigma}$. This motivates the contravariance of $f_{\Sigma}$: we have verified $e$ in the context of the resource $V$, but we intend to morph $e$ and execute it in a new resource context, where states are inhabitants of $W$.

More concretely, the precondition of morph $f \cdot e$ in Morph assumes $I s\_v$ and the existence of $s\_v = f_{\Sigma} s\_w$ such that $P s\_v$. By fault avoidance (i.e., type safety), a program (here $e$ in the premise) that is ascribed a Hoare type isn’t stuck. Hence, there exists a transition $t\_v$ by which $e$ steps from $s\_v$ into $s\_v'$. Because $I$ is an $f$-simulation, the commuting diagram implies the existence of $s\_w'$ such that $s\_v' = f_{\Sigma} s\_w'$ and $I s\_w'$. The morphing process is then iterated for $s\_w'$ and the subsequent states. This iteration relies on $f$ preserving $I$, much like a loop relies on the loop body preserving the loop invariant. Once $e$ terminates in a final state $s\_v''$, the postcondition in Morph must hold. First, $Q s\_v''$ must hold as $Q$ is $e$’s postcondition. Second, $I$ being an $f$-simulation yields the commuting diagram which implies the existence of $s\_w''$ such that $s\_v'' = f_{\Sigma} s\_w''$ (hence $f Q s\_w''$) and $I s\_w''$.

The paper can thus be seen as introducing simulations into separation logic in a simple\textsuperscript{4}, but also constructive manner. Customarily, an STS $W$ simulates another STS $V$ if whenever $V$ takes a transition, there exists a transition for $W$ to take [Lynch and Vaandrager 1995]. For us, a morphism $f$ computes the witness of this existential (via $f_{\Sigma}$), in the style of constructive logic and type theory. Moreover, a simulation is usually defined as a relation between the states of $V$ and $W$. For us, an $f$-simulation is a predicate on $W$-states alone, as $f_{\Sigma}$ deterministically computes the unique $V$-state that corresponds to a $W$-state in the simulation. Finally, simulations are also customarily required to relate a distinguished set of initial states of the source and target STSs. Our resources and $f$-simulations, on the contrary, don’t need to consider specific initial states of $V$ and $W$, because the initial states of any program are described by its precondition, and the Morph rule checks that the simulation holds on the pre-state of every invocation of morph.

The paper can also be seen as introducing a form of refinement mappings [Abadi and Lamport 1991] into separation logic, since refinement mappings, like morphisms, are functions between STSs. The two, however, have very different technical details, largely imposed by our connection to separation logic. This includes the introduction of the morphism action on programs and the Morph rule, but also the treatment of state ownership, ownership transfer, and framing (cf. Section 3), none of which have been considered with refinement mappings.

Moreover, resource morphisms go beyond mere program specification and proof, as they also support generic constructions over resources, such as “tensorsing” two resources, adjoining an invariant to a resource, or forgetting a ghost field from a resource (the last one with a mild generalization to indexed morphism families). Morphisms relate a construction to its components, much as arrows in category theory relate objects of universal constructions, and are thus essential for the constructions to compose. This is why we see resource morphisms as a step towards a general type-theoretic calculus of concurrent constructions.

We formalize the development in Coq, using Coq’s predicates over states as assertions\textsuperscript{5}, building on the code base of Fine-grained Concurrent Separation Logic (FCSL) [Nanevski et al. 2014]. The sources are available online as an Artifact [Nanevski et al. 2019a], along with the extended version of the paper [Nanevski et al. 2019b].\textsuperscript{6}

\textsuperscript{4}Our formalization exports nine rules for Hoare-style reasoning, each addressing an orthogonal linguistic feature.

\textsuperscript{5}As apparent from the Morph rule, we explicitly bind the state $s\_v$, in contrast to the classic presentation of separation logics where this state is implicit. This is merely a syntactic distinction.

\textsuperscript{6}In addition to the examples from the paper, the sources include further benchmarks such as Treiber stack [Treiber 1986], flat combiner [Hendler et al. 2010], a concurrent allocator, a concurrent graph spanning tree algorithm [Sergey et al. 2015a], ticketed [Mellor-Crummey and Scott 1991] and readers-writers [Courtois et al. 1971] locks.
Readmap. The rest of the paper is organized as follows. Section 2 introduces resources and associated notions of ghost state and transitions, via the spin lock example. Section 3 develops the theory formally, including our specific notion of framing. Section 4 illustrates how to morph spin locks into exclusive locks. Section 5 introduces indexed morphism families and applies them to “forgetting” the ghost state of a resource. This models what is often referred to as quiescence [Aspnes et al. 1994; Derrick et al. 2011; Jagadeesan and Riely 2014; Nanevski et al. 2014; Sergey et al. 2016], most commonly used when installing one concurrent structure into the private state of another. Section 6 discusses related work and Section 7 concludes.

2 BACKGROUND AND OVERVIEW

We illustrate our specification idiom, resources, and resource morphisms, by fleshing out the example of spin locks.

2.1 Histories

To specify the locking and unlocking methods over spin lock, we build on the idea of linearizability [Herlihy and Wing 1990], and record the operations on r in the linear sequence in which they occurred. We do so in Hoare triples, but in a thread-local way, i.e. from the point of view of the specified thread, which we refer to as “us” [Ley-Wild and Nanevski 2013].

Specifically, a program state s contains a ghost component that we project as τs s, and which keeps “our” history of lock operations. Dually, the projection τo s keeps the collective history of all “other” (i.e. environment) threads. Each thread has these two components in scope, but they may have different values in different threads. We refer to τs and τo as self and other histories, respectively [Nanevski et al. 2014; Sergey et al. 2015b].

A history is a timestamped log of the locking and unlocking operations. Mathematically, it’s a finite map from timestamps (strictly positive nats) to the set {L, U}. For example, the self history τs s defined as 2 ↦ U • 7 ↦ L • 9 ↦ L, signifies that “we” have unlocked at time 2, and locked at times 7 and 9. The timestamp gaps indicate the activity of the interfering threads, e.g., another thread must have locked at time 1, otherwise we couldn’t have unlocked at time 2. Similarly, another thread must have unlocked at time 8. The entries such as 2 ↦ U are singleton maps, and • is disjoint union, undefined if operand histories share a timestamp. We abbreviate by ˆs the history τs s • τo s, which is the combined history of all threads, and use Hist for the collection of all histories.

2.2 Resources

We next define the resource Spin from Section 1, that types spin lock methods. It is pictorially shown in Figure 3 on the left.

The state space of Spin, denoted Σ(Spin), makes explicit the assumptions about the components: that the histories are disjoint (denoted τs s ⊥ τo s), that the entries in ˆs alternate between L and U, and that r isn’t the null pointer.

The erasure ˆs s−3 shows how the state s maps to a heap once the ghost histories are removed. The expression r ↦ ω ( ˆs s) denotes a heap with only the pointer r, storing the Boolean value ω ( ˆs s). The latter computes the lock status out of the combined history ˆs; it equals true if the last log in the combined history is a lock entry L, and false otherwise.

The set of transitions of Spin, denoted ∆(Spin), contains lock_tr, unlock_tr, and the (elided) idle transition. The transition lock_tr adds a fresh L entry to τs s if ¬ω ( ˆs s), i.e., if the lock is free in the pre-state. Similarly, unlock_tr adds a fresh U entry if ω ( ˆs s), i.e., the lock is taken. The lock can by taken by “us” or by “others”, as ω is computed from the combined history ˆs. If the locking protocol insists that the thread that unlocks is the same thread that last locked, then the precondition of
We now give the following pidgin code for lock and unlock, intended to further the intuition about transitions. The actual implementation of the methods will be shown in Section 3, once we have formulated our system.

\[
\begin{align*}
\text{lock} & \equiv \text{do } (x \leftarrow \text{CAS}(r, \text{false}, \text{true}); \text{if } x \text{ then } \tau_x s := \tau_x s \cdot \text{fresh}(\hat{r}s) \Rightarrow L); \text{while } \neg x \\
\text{unlock} & \equiv (x \leftarrow !r; \text{r := false}; \text{if } x \text{ then } \tau_x s := \tau_x s \cdot \text{fresh}(\hat{r}s) \Rightarrow U)
\end{align*}
\]

The brackets (\( \langle \rangle \)) denote atomic execution (i.e., uninterrupted by other threads) of real and ghost code, the latter given in gray. Note how the bracketed code in lock implicitly describes a choice, depending on the contents of \( r \), between executing lock_tr or the idle transition in the resource. The former, when considered on erased states, corresponds to CAS successfully setting \( r \), the latter to CAS failing. Similarly, unlock chooses between unlock_tr and the idle transition. Thus, we shall abstractly view the atomic executions as a choice between transitions of the corresponding resource, rather than as bracketing of ghost with real code.

We can now explain the history-based specs for lock and unlock.

\[
\begin{align*}
\text{lock} : [h, k].\{\lambda s. \tau_s s = h \land k \leq \text{last\_stamp}(\hat{r}s)\} \\
& \quad \{\lambda s. \exists t. \tau_s s = h \Rightarrow L \land k < t\} @\text{Spin} \\
\text{unlock} : [h, k].\{\lambda s. \tau_s s = h \land k \leq \text{last\_stamp}(\hat{r}s)\} \\
& \quad \{\lambda s. \exists t. \tau_s s = h \Rightarrow U \land k < t \lor \tau_s s = h \land \hat{r}s t = U \land k \leq t\} @\text{Spin}
\end{align*}
\]
The precondition of lock starts with self history $\tau_s s$ equal to $h^7$, which is increased in the postcondition to log a locking event at time $t$. The conjunct $k < t$ in the postcondition claims that $t$ is fresh, because it’s larger than any $k$ generated prior to the call (as $k \leq \text{last\_stamp}(\hat{r} s)$ is a conjunct in the precondition, and $k$ is universally quantified outside of the pre- and postcondition). The natural numbers ordering on timestamps gives the linear sequence in which the events logged in $\hat{r} s$ occurred. Notice that the spec is stable, i.e., invariant under interference. Intuitively, other threads can’t modify the $\tau_s s$ field, as it’s private to “us”. They can log new events into $\tau_o s$, which features in the comparison $k \leq \text{last\_stamp}(\hat{r} s)$, but this only increases the right-hand side of the comparison and doesn’t invalidate it.

Similarly, unlock starts with history $h$, which is either increased to log a fresh unlocking event at time $t$, or remains unchanged if the unlocking fails because unlock encounters $r$ already freed at time $t$ (conjunct $\hat{r} s t = U$). The conjunct $k \leq t$ captures that another thread may have freed $r$ after the invocation of unlock ($k < t$), or that we invoked unlock with $r$ already freed ($k = t$).

Observe that the spec for unlock doesn’t require that the unlocking thread is the one that last locked, or even that the lock is taken when unlocking is attempted. This is so because we intend the specs to capture only the basic mechanics of spin locks, and leave it to the clients to supply application-specific policies, via morphing, as we illustrate on exclusive locks in Section 4 (and on readers-writers locks in [Nanevski et al. 2019a]).

2.4 Morphisms

Consider next how to express a client of lock that, simultaneously with a successful lock, adds $n$ to the ghost component $\kappa_s s$ of resource Counter (right half of Figure 3). Intuitively, we desire something like $\langle \text{lock}; \ k_s s := \kappa_s s + n \rangle$ but this isn’t quite right. Indeed, bracketing would prevent other programs from running during the iterations of lock’s loop, thus changing the granularity of the program. We want to model that addition occurs only upon the successful CAS of the last iteration in lock. To do so, we use morphisms as follows.

First, we “tensor” the resources Spin and Counter, as graphically indicated8 in Figure 3; that is, we create a new resource SC whose state is a pair of Spin and Counter states, and transitions are lock_tr $\Leftrightarrow$ incr_tr $n$ and unlock_tr $\Leftrightarrow$ id_tr. Operator $\Leftrightarrow$ (pronounced “couple”) indicates that the operand transitions are executed simultaneously on their respective state halves. It’s defined as follows, where $s\backslash 1$ and $s\backslash 2$ project state $s$ to its Spin and Counter components, respectively:

$$(t_1 \Leftrightarrow t_2) s s' \equiv t_1 (s\backslash 1) (s'\backslash 1) \wedge t_2 (s\backslash 2) (s'\backslash 2)$$

Second, for each $n \in \mathbb{N}$, we define the morphism $f_n : \text{Spin} \rightarrow \text{SC}$ as follows:

$$(f_n)\exists s s' \equiv s\backslash 1$$

$$(f_n)\wedge s \text{ lock}_\text{tr} \equiv \text{lock}_\text{tr} \Leftrightarrow \text{incr}_\text{tr} n$$

$$(f_n)\wedge s \text{ unlock}_\text{tr} \equiv \text{unlock}_\text{tr} \Leftrightarrow \text{id}_\text{tr}$$

This definition captures: (1) starting from an SC state $s$, we can obtain a Spin state by taking the first projection; (2) a Spin program can be lifted to SC by changing the transition lock_tr by $(f_n)\wedge$ on the fly, to increment $\kappa_s s$ simultaneously with the lock acquisition; and (3) unlock_tr is coupled with the idle transition in Counter, thus $\kappa_s s$ is unchanged by unlocking.

Now, our desired program is

$$\text{morph } f_n \text{ lock}$$

---

7As customary in Hoare logic, $h$ and $k$ are logical variables, used to relate the pre and post-state. They are universally quantified, scoping over pre and post-condition, and the syntax $\dots$ makes the binding explicit.

8We elide the definition of tensoring, as it isn’t required to follow the presentation. It can be found in the online appendices [Nanevski et al. 2019b, Appendix A].
The proof outline in Figure 4 shows that $\kappa_s$ $s$ increments by 3, $\#_L$ ($\cdot$) is the number of L-entries in a history.

which is typed by SC, and executes lock_tr $\Rightarrow$ incr_tr $n$ whenever lock executes lock_tr, thus incrementing $\kappa_s$ $s$ precisely, and only, upon a successful CAS.

### 2.5 Inference

The Morph rule provides a way to reason about morphed programs. To illustrate the proofs, we consider the following simple program

```
 morph $f_1$ lock;
 morph $f_2$ unlock;
 morph $f_2$ lock
```

which, in addition to locking and unlocking, increments $\kappa_s$ $s$ by 1 in the first line, and by 2 in the third line. The second line morphs unlock vacuously, as unlocking leaves $\kappa_s$ $s$ unchanged. Nevertheless, some morphing of unlock is necessary, to bring the commands under the same resource type.

The proof outline in Figure 4 shows that $\kappa_s$ $s$ increments by 3, and we discuss its main points next. In the outline, $\#_L$ is a function on history that computes the number of L entries in the history. The outline starts with the precondition $\kappa_s$ $s$ = $n$, where $n$ snapshots "our" current count. Line 2 uses $h$ to snapshot "our" history. Line 3 applies morph to lock, and correspondingly, the Morph rule in the proof. At this point, we choose the simulation $L_1$ as indicated in line 3, to state that the counter $\kappa_s$ increments $n$ by the number of fresh L-entries in the history. Intuitively, $L_1$ is an $f_1$-simulation because it is preserved under incrementing $\kappa_s$ $s$ by 1 while simultaneously adding an L-entry to $\tau_s$ $s$ (Figure 5). It’s easy to see that $L_1$ holds in line 2, thus by Morph, it holds in line 4 as well. But, in line 4, by postcondition of lock, the history $\tau_s$ $s$ has one more locking entry. Thus, $\kappa_s$ $s$ is increased by 1 (line 5). The remainder of the outline proceeds similarly.

We close the discussion with the observation that the property of being a simulation (i.e., making diagrams in Figures 2 and 5 commute) relies only on the resource in the program’s type, and the morphism in question, not on the program’s code, as required for compositional reasoning. In this respect, the simulations are different from loop invariants, which are properties of programs. The Morph rule ties the simulations to the morphed program by conjoining them with the program’s pre- and the postcondition. Specifically above, $L_1$ enables computing the end-value of $\kappa_s$ $s$ from the end-value of $\tau_s$ $s$, and $\tau_s$ $s$ is given by the spec of lock.

---

9Strictly speaking, we should write $\kappa_s$ $(s\backslash 2)$ (resp. $\tau_s$ $(s\backslash 1)$) to extract the self component of the Counter (resp. Spin) "sub-resource" of SC. However, the components have different names, so there’s no confusion which projection of $s$ they come from. We thus abbreviate $\kappa_s$ $(s\backslash 2)$ with $\kappa_s$ $s$, $\tau_s$ $(s\backslash 1)$ with $\tau_s$ $s$, and similarly for $\kappa_o$ and $\tau_o$. 

3 DEFINITIONS OF THE FORMAL STRUCTURES

To develop the notions of morphisms and simulations, we first require a number of auxiliary definitions, such as states, transitions, and resources on which morphisms act. This section defines all the concepts formally, culminating with the inference rules of our system.

3.1 States

3.1.1 Subjective Components. Different resources may contain different state components, e.g., \( \tau \) of Spin and \( \kappa \) of Counter. In general, a state is parametrized by two types: \( M \) classifies the self and other components, and \( T \) classifies the joint (aka., shared) state. Thus, \( s = (a_s, a_j, a_o) \) is a state if \( a_s, a_o \in M \), and \( a_j \in T \). If we want to be explicit about the types, we say that \( s \) is an \((M, T)\)-state. We use \( a_s, a_j \) and \( a_o \) as generic projections out of \( s \), but rename them in specific cases, for readability. For example, in the case of Spin: \( M \) is Hist, \( T \) is unit type, and \( \tau_s/\tau_o \) renames \( a_s/a_o \). In the case of Counter: \( M \) is \( \mathbb{N} \), \( T \) is unit type, and \( \kappa_s/\kappa_o \) renames \( a_s/a_o \).

Because \( a_s \) and \( a_o \) represent thread-specific views of the state, we refer to them as subjective components, and to \( s \) as subjective state [Ley-Wild and Nanevski 2013].

3.1.2 Algebra of Subjectivity. The specs must often combine the subjective components, cf. how histories were unioned by \( \bullet \) to express timestamp freshness in the spec of lock. To make the combination uniform, \( M \) is endowed with the structure of a partial commutative monoid (PCM). A PCM is triple \((M, \bullet, 1)\) where \( \bullet \) (join) is a partial, commutative, associative, binary operation on \( M \), with \( 1 \) as the unit. As a generic notation, we write \( x \perp y \) to denote that \( x \bullet y \) is defined.

Example PCMs are Hist with disjoint union and the empty history \( \emptyset \), and \( \mathbb{N} \) with + and 0. Another common PCM is the set of heaps (denoted Heap). Heaps map pointers to values, and are thus similar to histories, which map timestamps to operations. We can therefore reuse the history notation, and write, e.g.:

\[
x \mapsto 3 \bullet y \mapsto \text{false}
\]

to describe the heap containing pointers \( x \) and \( y \), storing 3 and false, respectively.\(^{10}\) Heap is a PCM with disjoint union and the empty heap \( \emptyset \), similar to Hist. Cartesian product of PCMs is a PCM, so PCMs can be combined, cf. the PCM of SC is constructed out of those of Spin and Counter in Section 2.

3.1.3 Subjectivity and Parallel Composition. The subjective components are local, in the sense that they have different values in different threads. However, despite the locality, the components of different threads aren’t independent, but are inter-related as shown in Figure 6.

Imagine three threads \( \theta_1, \theta_2 \) and \( \theta_3 \) running concurrently. Their respective states must have the forms \( s_1 = (a_1, a_j, a_2 \bullet a_3), s_2 = (a_2, a_j, a_3 \bullet a_1) \) and \( s_3 = (a_3, a_j, a_1 \bullet a_2) \). Indeed, any two of the threads combined are the environment for the third thread. Thus, the PCM join of the self’s of any

\(^{10}\)We silently already used this notation to define the erasure function for Spin in Figure 3.
two threads must equal the other of the third thread. Figures 6(1) and 6(2) illustrate this property for threads θ₁ and θ₂, with θ₃ being their implicit environment.

If θ is the parent thread of θ₁ and θ₂, then its state is s = (a₁ * a₂, a₃), since θ is the combination of θ₁ and θ₂, and has θ₃ as the environment. We abbreviate as s = s₁ * s₂ the relationship between the parent state s, and the children states s₁ and s₂, and illustrate it in Figure 6(3).

3.1.4 Globality. A property or a function is global if it remains invariant under moving PCM values between subjective components. In light of Figure 6, such properties and functions obtain equal valuations across all concurrent threads, thus justifying the name. We introduce several operations for surgery on subjective states, and then use them to define globality and conditional globality, where the invariance holds only under a (global) condition.

Definition 3.1. Let p ∈ M and s = (a₅, a₆, a₇) be an (M, T)-state. The self-framing of s with the frame p is the state s □ p = (a₅ * p, a₆, a₇). Dually, the other-framing of s with p is the state s ▽ p = (a₅, a₆, p * a₇).

Definition 3.2. A predicate P is global, if P (s ▽ p) ⇔ P (s □ p) for every p and state s such that a₅ s ⊥ p ⊥ a₇ s. A (partial) function f on states is global, if f (s ▽ p) = f (s □ p) under the same conditions.

Examples of global predicates from Section 2 are P s = τ₅ s ⊥ τ₇ s, and Q s = alternate (τ s) used in Figure 3 to characterize Spin histories. P and Q are both defined in terms of τ s = τ₅ s * τ₇ s; Q directly so, and P because τ₅ s ⊥ τ₇ s iff τ₅ s • τ₇ s is itself defined. In other words, both P and Q express a property of the collective history of all threads operating over Spin, taken together. Clearly, the value of this history is invariant across all the threads, and therefore, so are P and Q. Specifically, they are invariant under shuffling timestamps between τ₅ and τ₇, as this doesn’t alter the total. In fact, τ itself is a global function, so we proceed to refer to τ as the global history.

Definition 3.3. Let X be a global predicate. A predicate P is global under X, if P (s ▽ p) ⇔ P (s □ p) for every p and state s such that a₅ s ⊥ p ⊥ a₇ s and X (s ▽ p). Similarly for functions.

3.1.5 Subjectivity and Framing. Subjective state makes framing work somewhat differently than in the customary, non-subjective, separation logics. The latter may be viewed as having the self component, but lacking other. To illustrate the difference, we give lock (Section 2) the following spec, which is small [O’Hearn et al. 2001] wrt. the history τ₅ s,

\[
\text{lock : } [k]. \{ \lambda s. \tau s s = \emptyset \land k \leq \text{last\_stamp}(\tau o s) \} \\
\{ \lambda s. \exists t. \tau s s = t \implies L \land k < t \} @\text{Spin}
\]

and then we frame the history h onto τ₅ s to obtain the equivalent large spec we actually presented:

\[
\text{lock : } [h, k]. \{ \lambda s. \tau s s = h \land k \leq \text{last\_stamp}(\hat{r} s) \} \\
\{ \lambda s. \exists t. \tau s s = h \land k < t \} @\text{Spin}
\]
As expected in separation logic, framing increased the starting \(\tau_0 s\) from \(\emptyset\) to \(h\), which is the key distinction between small and large specs. But this isn’t all it did; it also deducted \(h\) from \(\tau_0 s\). Indeed, had \(\tau_0 s\) been unchanged (as might also be expected in separation logic), then both specs would contain the same conjunct \(k \leq \text{last\_stamp}(\tau_0 s)\). But the large spec contains \(k \leq \text{last\_stamp}(\hat{\tau} s) = \text{last\_stamp}(h \bullet \tau_0 s)\), where \(h\) is joined to \(\tau_0 s\) to compensate for the deduction.

To explain the deduction, notice that in any separation logic, framing is a special case of parallel composition. To add a frame \(h\) to the state of a program \(e\), it suffices to compose \(e\) in parallel with the idle thread having \(h\) as its self. The composition executes like \(e\), but with self enlarged by \(h\), and \(h\) remains unchanged. In the subjective setting, parallel composition joins the self’s of two threads, but also decreases the other of the parent, as illustrated in Figure 6. It is this decrease that is evidenced in the large spec.

Therefore, framing enlarges self by \(h\), and simultaneously removes \(h\) from other, which must already contain \(h\). Framing shuffles existing state between components, but doesn’t introduce new state, in contrast to the usual separation logic formulations. This preserves the values of global functions, and facilitates their use in specs (e.g., the global history \(\hat{\tau}\) in lock).

### 3.2 Resources

Resources consist of state spaces and transitions. The state spaces describe the properties that hold for all threads of the resource, so we use global predicates and functions to represent them.

**Definition 3.4.** A **state space** is a pair \(\Sigma = (P, \tau^{-\top})\), where \(P\) is a global predicate and \(\tau^{-\top}\) is a partial function into heaps, global under \(P\), called erasure, such that for every state \(s\), \(P s\) implies \(a_s(s) \subseteq a_o(s)\) and \(\tau^{-\top} s\) is defined. We write \(s \in \Sigma\) to mean \(P s\).

Transitions describe the allowed atomic modifications on state. We require the following properties of them, to facilitate separation-style reasoning.

**Definition 3.5.** A **transition** \(t\) over state space \(\Sigma\) is a binary relation on \(\Sigma\) states, such that:

1. (partial function) if \(t s s'\) and \(t s' s''\) then \(s' = s''\).
2. (other-fixity) if \(t s s'\), then \(a_o s = a_o s'\).
3. (transition locality) if \(t (s \triangleright p) x\), then there exists \(s'\) such that \(x = s' \triangleright p\) and \(t (s \triangleleft p) (s' \triangleleft p)\)

A state \(s\) is **safe** for \(t\), if there exists \(s'\) such that \(t s s'\).

Property (2) captures that a transition can’t change the other-component, as it’s private to other threads. However, a transition can read this component, cf. how lock_tr in Figure 3 uses \(\tau_0 s\) as part of \(\hat{\tau} s\) to compute a fresh timestamp.

Transition locality (3) essentially says that transitions can be framed. To see how, let \(\theta_1\) be a thread in the state \(s \triangleright p\), whose sibling \(\theta_2\) has self-component \(p\). Their parent \(\theta\) is thus in the state \(s \triangleleft p\), by Figure 6. If \(\theta_1\) performs a transition \(t (s \triangleright p) x\), then by (3), the move can be seen as a transition of \(\theta\) in the state \(s \triangleleft p\). In other words, the transition of a child can be seen as a transition of the parent, but with self enlarged by \(p\), and other suitably reduced by \(p\). This is precisely the view of framing described in Section 3.1.5. Hence, transition locality is the base case of, and gives rise to, framing on programs, as a program’s execution is a sequence of transitions.

**Definition 3.6.** A \(\Sigma\)-transition \(t\) is **footprint preserving** if \(t s s'\) implies that \(\tau^{-\top} s\) and \(\tau^{-\top} s'\) contain the same pointers.

Transitions that preserve footprints are important because they can be coupled with other such transitions without imposing side conditions on the combination. For example, consider the incr_tr transition of Counter in Figure 3, which is footprint preserving, as it doesn’t allocate or deallocate any pointers. Were it also to allocate, we will have a problem when combining Spin and Counter, as
we must impose that incr_tr won’t allocate the pointer r, already taken by Spin. For simplicity, we here present the theory with only footprint-preserving transitions, but have added non-preserving (aka. external) transitions as well [Nanevski et al. 2019a]. External transitions encode transfer of data in and out of a resource [de Alfaro and Henzinger 2001], of which allocation and deallocation are an instance. When a resource requires allocation or deallocation, it can be tensored with an allocator resource to exchange pointers through ownership transfer [Filipović et al. 2010b; Nanevski et al. 2014] via external transitions. We elide further discussion, but refer to the Coq files for the implementation of an allocator resource and example programs that use it.

**Definition 3.7.** A resource is a tuple $V = (M, T, \Sigma, \Delta)$, where $\Sigma$ is a space of $(M, T)$-states, and $\Delta$ a set of footprint preserving $\Sigma$ transitions. We refer to $V$’s components as projections, e.g. $\Sigma(V)$ for the state space, $\Delta(V)$ for the transitions, $M(V)$ for the PCM, etc. A state $s$ is $V$-state iff $s \in \Sigma(V)$.

We close the discussion on resources by defining actions—atomic operations on (combined real and ghost) state, which are the basic building blocks of programs.

**Definition 3.8.** An action of type $A$ in a resource $V$ is a partial function $a : \Sigma(V) \to \Delta(V) \times A$, mapping input state to output transition and value, which is local, in the sense that it is invariant under framing. Formally, if $a(s \triangleright p) = (t, v)$ then $a(s \triangleleft p) = (t, v)$; that is, if $a$ is performed by a child thread, it behaves the same when viewed by the parent.

The effect of $a$ is the partial function $[a] : \Sigma(V) \to \Sigma(V) \times A$ mapping input state to output state and value, defined as $[a]s = (s', v)$ iff $\exists t. a(s = (t, v))$. Note that $[a]$ is a (partial) function because $a$ and $t$ are.

For example, we model the bracketed code used in the lock loop in Section 2, as the following action of type bool:

$$
\begin{align*}
\text{trylock\_act}_s &\equiv \begin{cases} 
(\text{lock\_tr}, \text{true}) & \text{if } \neg \omega(\hat{s}) \\
(\text{id\_tr}, \text{false}) & \text{otherwise (1)}
\end{cases}
\end{align*}
$$

The action is local, as it depends only on $\hat{s}$, which is invariant under framing.

We say that a erases to an atomic read-modify-write (RMW) command $c$ [Herlihy and Shavit 2008], if $[a]$ behaves like $c$ when the states are erased to heaps. In other words, if $[a]s = (s', v)$, then $c \triangleright s'^3 = (c \triangleright s'^3, v)$. One may check that trylock\_act erases to CAS(r, false, true), as expected.\footnote{All the actions we use in this paper and in the Coq code erase to some RMW command. However, we proved this only by hand, as our formalism and the Coq implementation don’t currently issue proof obligations to check this. In general, we currently treat code and ghost code equally, and, as customary in type theory, equally to proofs. Differentiating between these formally is an orthogonal issue that we plan to address in the future by making a type distinction between them, such as in the work on proof irrelevance in type theory [Barras and Bernardo 2008; Gilbert et al. 2019; Pfenning 2001].}

Similarly,

$$
\begin{align*}
\text{unlock\_act}_s &\equiv \begin{cases} 
(\text{unlock\_tr}, ()) & \text{if } \omega(\hat{s}) \\
(\text{id\_tr}, ()) & \text{otherwise (2)}
\end{cases}
\end{align*}
$$

is an action of unit type, which erases to $r := \text{false}$.

### 3.3 Morphisms

**Definition 3.9.** A resource morphism $f : V \to W$ consists of two partial functions $f_\Sigma : \Sigma(W) \to \Sigma(V)$ (note the contravariance), and $f_\Delta : \Delta(W) \to \Delta(V)$, such that:

1. (locality of $f_\Sigma$) there exists a function $\phi : M(W) \to M(V)$ such that if $f_\Sigma(s'_w \triangleright p) = s'_v$, then there exists $s'_v$ such that $s'_v = \phi(p)$, and $f_\Sigma(s'_w \triangleleft p) = s'_v \triangleleft \phi(p)$.
2. (locality of $f_\Delta$) if $f_\Delta(s'_w \triangleright p)(t_u) = t_w$, then $f_\Delta(s'_w \triangleleft p)(t_u) = t_w$.
3. (other-fixity) if $a_o(s'_w) = a_o(s'_w')$ and $f_\Sigma(s'_w)$, $f_\Sigma(s'_w')$, exist, then $a_o(f_\Delta(s'_w)) = a_o(f_\Delta(s'_w'))$. 

11
A morphism \( f \) transforms a \( V \)-program \( e \) into a \( W \)-program, as follows. When morph \( f \) e is in a \( W \)-state \( s_w \), it has to determine a \( W \)-transition to take. It does so by obtaining a \( V \)-state \( s_v = f_\Sigma (s_w) \). Next, out of \( s_v, e \) can determine the transition \( t_v \) to take. The morphed \( W \)-program then takes the \( W \)-transition \( f_\Lambda (s_w)(t_v) \).

The properties (1) and (2) of Definition 3.9 provide basic technical conditions for this process to be invariant under framing. Property (1) is a form of "simulation of framing", i.e., a frame \( p \) in \( W \) can be matched with a frame \( \phi (p) \) in \( V \). Thus, framing a morphed program can be viewed as framing the original program. Property (2) says that framing doesn’t change the transition that \( f_\Lambda \) produces; thus it doesn’t influence the behavior of morphed programs. The property (3) restricts the choice of \( s_v' \) in (1) so that \( a_o (s_v') \) is uniquely determined by \( a_o (s_w) \), much as how \( \phi (p) \) in (1) is uniquely determined by \( p \). This is a technical condition which we required to prove the soundness of the frame rule.

Example. Properties (1)-(3) are all satisfied by the morphisms \( f_n : \text{Spin} \rightarrow \text{SC} \) from Section 2. Indeed, \( M (\text{SC}) = M (\text{Spin}) \times M (\text{Counter}) = \text{Hist} \times \mathbb{N} \). Thus, a frame in \( \text{SC} \) is a pair of a history and a nat; it is transformed into a frame in \( \text{Spin} \) just by taking the history component. We thus instantiate \( \phi \) in (1) with the first projection function, and it is easy to see that it satisfies the rest of (1). Property (2) holds because \( (f_n)_\Lambda \) doesn’t depend on the state argument, hence framing this state doesn’t change the output. Finally, in (3), the values \( a_o (s_w) \) and \( a_o (s_v') \) are also pairs of a history and a nat. If the pairs are equal, then their history components are equal too, deriving (3).

Finally, resources and their morphisms support a basic categorical structure, under the following notions of morphism identity and composition. We have proved in the Coq files that morphism composition is associative, with the identity morphism as the unit, where two morphisms are equal if their \( \Sigma \) and \( \Lambda \) components are equal as partial functions.

Definition 3.10. The identity morphism \( \text{id} : V \rightarrow V \) is defined by \( \text{id}_\Sigma s = s \) and \( \text{id}_\Lambda s t = t \). The composition of morphisms \( f : U \rightarrow V \) and \( g : V \rightarrow W \) is the morphism \( g \circ f : U \rightarrow W \) defined by:

\[
(g \circ f)_\Sigma s_w \equiv f_\Sigma (g_\Sigma s_w) \\
(g \circ f)_\Lambda s_w t_u \equiv g_\Lambda s_w (f_\Lambda (g_\Sigma s_w) t_u)
\]

3.4 Simulations

Because \( f_\Sigma \) and \( f_\Lambda \) are partial, a program lifted by a morphism isn’t immediately guaranteed to be safe (i.e., doesn’t get stuck). For example, the state \( s_w = f_\Sigma s_w \), whose computation is the first step of morphing, needn’t exist. Even if \( s_w \) does exist, and the original program takes the transition \( t_v \) in \( s_v \), then \( t_w = f_\Lambda s_w t_v \) needn’t exist. Even if \( t_w \) does exist, there is no guarantee that \( s_w \) is safe for \( t_w \). An \( f \)-simulation is a condition that guarantees the existence of these entities, and their mutual agreement (e.g., that \( s_w \) is safe for \( t_w \)), so that a morphed program that typechecks against the \text{Morph} rule doesn’t get stuck.

Definition 3.11. Given a morphism \( f : V \rightarrow W \), an \( f \)-simulation is a predicate \( I \) on \( W \)-states such that:

1. If \( I s_w \), and \( s_v = f_\Sigma (s_w) \) exists, and \( t_v s_v s_v' \), then there exist \( t_w = f_\Lambda s_w t_v \) and \( s_w' \) such that \( I s_w', t_v s_w s_w' \).

2. If \( I s_w \), and \( s_v = f_\Sigma (s_w) \) exists, and \( s_w \rightarrow^* s_v ' \), then \( I s_w \), and \( s_v = f_\Sigma (s_w) \) exists, and \( s_v \rightarrow^* s_v' \). Here, the relation \( s \rightarrow^* s' \) denotes that \( s \) other-steps by \( W \) to \( s' \), i.e., that there exists a transition \( t \in \Delta (W) \) such that \( t s \rightarrow^* s' \). The transposition \( s^\top = (a_o s, a_j s, a_s s) \) swaps the subjective components of \( s \), to obtain the view of other threads. The relation \( \rightarrow^* \) is the reflexive-transitive closure of \( \rightarrow \), allowing for an arbitrary number of steps.
Property (1) says that $W$ simulates $V$ on states satisfying $I$. Property (2) states the simulation in the opposite direction, i.e., of $W$ by $V$, but allowing many other-steps to match many other-steps. Notice that other-stepping transitions over transposed states; that is, it changes the other, but, by Definition 3.5(2), preserves the self of the states. Intuitively, (2) ensures that interference in $W$ may be viewed as interference in $V$, so that stable Hoare triples in $V$ can be transformed into stable Hoare triples in $W$, which is required for the soundness of the Morph rule. Property (1) has already been shown in Figure 2; we repeat it in Figure 7, together with a diagram for property (2).

### 3.5 Inference Rules

We present the system using the Calculus of Inductive Constructions (CiC) as an environment logic, hence as a shallow embedding in Coq. We inherit from CiC the useful concepts of higher-order functions and substitution principles, and only present the notions specific to Hoare logic.\footnote{The extended version of the paper [Nanevski et al. 2019b, Appendix D] defines the denotational semantics, in CiC, for these notions, and states a theorem, proved in Coq, that the inference rules are sound wrt. the denotational semantics.}

We differentiate between two different notions of program types: ST $V A$ and $[\Gamma]$. $\{P\} A \{Q\}@V$. The first type circumscribes programs that respect the transitions of the resource $V$, and return a value of type $A$ if they terminate. The second, Hoare type, is a subset of ST $V A$, selecting only those programs that satisfy the precondition $P$ and postcondition $Q$, under the context $\Gamma$ of logical variables. To accommodate for the return values, the postcondition $Q$ is now a predicate over values of type $A$ and states (if $A = \text{unit}$, we elide it from the Hoare type, as we did in Section 2).

The key concept in the inference rules is the predicate transformer $vrf e Q$, which takes a program $e : ST V A$, and a postcondition $Q$, and returns the set of $V$-states from which $e$ is safe to run\footnote{Thus ensuring fault avoidance.} and produces a result $v$ and ending state $s'$ such that $Q v s'$. Hoare types are then defined in terms of $vrf$, as follows.

$$[\Gamma]. \{P\} A \{Q\}@V = \{e : ST V A \mid \forall \Gamma. \forall s \in \Sigma(V). P s \rightarrow vrf e Q s\}$$

We formulate the system using both $vrf$ and the Hoare types. The former is useful, as it leads to compact presentation, avoiding a number of structural rules of Hoare logic. The latter is useful because it lets us easily combine Hoare reasoning with higher-order concepts. For example, having inherited higher-order functions from CiC, we can immediately give the following type to the fixed-point combinator, where $T$ is the dependent type $T = \Pi_{x:A}. [\Gamma]. \{P\} B \{Q\}@V$:

$$\text{fix} : (T \rightarrow T) \rightarrow T$$

Fix

Here, $T$ serves as a loop invariant; in $\text{fix}(\lambda f. e)$ we assume that $T$ holds of $f$, but then have to prove that it holds of $e$ as well, i.e., it is preserved upon the end of the iteration.

In reasoning about programs, we keep the transformer $vrf$ abstract, and only rely on the following minimal set of rules. These, together with the above definition of Hoare types and typing for $\text{fix}$,
are the only Hoare-related rules of the system. In the rules we assume that $e : ST V A$, $e_i : ST V A_i$, $a$ is a $V$-action, $f : V \to W$ is a morphism, $I$ is an $f$-simulation, $s \in \Sigma(V)$, and $s_w \in \Sigma(W)$.

\[
\begin{align*}
\text{vrf\_post} & : (\forall v s. J s \to Q_1 v s \to Q_2 v s) \to J s \to \text{vrf} e Q_1 s \to \text{vrf} e Q_2 s \\
\text{vrf\_ret} & : (Q v) s \to \text{vrf} (\text{ret v}) Q s \\
\text{vrf\_bnd} & : \text{vrf} e_1 (\lambda x. \text{vrf} (e_2 x) Q) s \to \text{vrf} (x \leftarrow e_1; (e_2 x)) Q s \\
\text{vrf\_par} & : ((\text{vrf} e_1 Q_1) \ast (\text{vrf} e_2 Q_2)) s \to \text{vrf} (e_1 \parallel e_2) (\lambda v. (Q_1 v) \ast (Q_2 v)) s \\
& \quad \text{where } (P \ast Q) s \equiv \exists s_1 s_2. s = s_1 \ast s_2 \land P s_1 \land Q s_2 \\
\text{vrf\_frame} & : ((\text{vrf} e_1 Q_1) \ast Q^* s) \to \text{vrf} e (\lambda v. (Q_1 v) \ast Q_2) s \\
\text{vrf\_act} & : (\lambda s'. \exists s'' v. [a] s' = (s'', v) \land (Q v) s'' \ast s \to \text{vrf} (a) Q s \\
\text{vrf\_morph} & : f^*(\text{vrf} e Q) s_w \to I s_w \to \text{vrf} (\text{morph f} e) (\lambda v. s'_w, f^*(Q v) s'_w \ast I s'_w) s_w \\
& \quad \text{where } f R s_w \equiv \exists s_{UV}. s = f s_w \land R s_U 
\end{align*}
\]

In English:

- The vrf\_post rule weakens the postcondition, similar to the well-known rule of Consequence in Hoare logic. The rule allows assuming a property $J$ when establishing a postcondition $Q_2$ out of $Q_1$. Here $J$ is an invariant, i.e., a property preserved by the transitions of $V$; an id-simulation. Thus, invariants can be elided from program specs, and invoked by vrf\_post when needed.
- The vrf\_ret rule applies to an idle program returning $v$. When we want an idle program that returns no value, we simply take $v$ to be of unit type. The rule explicitly stabilizes the postcondition $Q$ to allow for the state $s$ to be changed by interference of other threads in between the invocation of the idle program and its termination. Here, stabilization of a predicate $Q$ is $Q^*(s) \equiv \forall s'. s \to^* s' \to Q(s')$. The predicate $Q$ is stable if $Q = Q^*$, and it is easy to see that $Q^*$ is stable for every $Q$.
- The vrf\_bnd rule is a Dijkstra-style rule for sequential composition. In order to show that the sequential composition $x \leftarrow e_1; (e_2 x)$ has a postcondition $Q$, it suffices to show that $e_1$ has a postcondition $\lambda x. \text{vrf} (e_2 x) Q$. In other words, $e_1$ terminates with a value $x$ and in a state satisfying $\text{vrf} (e_2 x) Q$, so that running $e_2 x$ in that state yields $Q$.
- The vrf\_par and vrf\_frame rules are predicate transformer variants of the rules for parallel composition and framing from separation logic. The separating conjunction $P \ast Q$ is defined as customary in separation logic, except that we use the subjective splitting of state, as explained in Section 3.1.3 and Figure 6. The vrf\_frame rule can be seen as an instance of vrf\_par, where $e_2$ is taken to be the idle programs returning no value. Thus, $Q_2$ is explicitly stabilized in vrf\_frame, to match the precondition of the vrf\_ret rule for idle programs.
- The vrf\_act rule says that $Q$ holds after executing action $a$ in state $s$, if $s$ steps to $s'$ by interfering threads, and then $[a] s'$ returns the pair $(s'', v)$ of output state and value. The latter satisfy the stabilization of $Q$, to allow for interference on $s''$ after the termination of $a$.
- The vrf\_morph rule is a straightforward casting of the $\text{Morph}$ rule from Section 1 into a predicate transformer style.

Finally, we also inherit all the CiC logical and programming constructs as well, which has important consequences for Hoare-style reasoning. For example, in CiC one can form conditionals over any type, including propositions and $ST V A$ types. Thus, given a Boolean $b$ and $e_1, e_2 : ST V A$, the following rule, derivable by case analysis on $b$, allows us to write programs that use conditionals, and verify them in the usual Hoare-style logic.

\[
\text{vrf\_cond} : (\text{if } b \text{ then } \text{vrf} e_1 Q s \text{ else } \text{vrf} e_2 Q s) \to \text{vrf} (\text{if } b \text{ then } e_1 \text{ else } e_2) Q s
\]

All the other customary rules of Hoare logic also become derivable. For example, if $e : \{P\} \{Q\}$ and $\forall s \in \Sigma(V). P' s \to P s$, then also $e : \{P'\} \{Q\}$. Similarly, if $e$ depends on a logical variable $x : A$
tyLck ≡ [h, k]. \{λs. τₗ s = h ∧ k ≤ last_stamp (τ s)\} \{λs. \exists t. τₗ s = h • t \implies L ∧ k < t\}@Spin

1. fix (λloop : unit→tyLck. λ_: unit.
2. \{τₗ s = h ∧ k ≤ last_stamp (τ s)\}
3. \{∃s’ b. [trylock_act] s = (s’, b) ∧
   if b then ∀t. τₗ s’ = h • t \implies L ∧ k < t else τₗ s’ = h ∧ k ≤ last_stamp (τ s’)\}
4. b ← (trylock_act);
5. \{if b then ∀t. τₗ s = h • t \implies L ∧ k < t else τₗ s = h ∧ k ≤ last_stamp (τ s)\}
6. if b then \{∃t. τₗ s = h • t \implies L ∧ k < t\} ret () \{∃t. τₗ s = h • t \implies L ∧ k < t\}
7. else \{∀t. τₗ s = h ∧ k ≤ last_stamp (τ s)\} loop () \{∃t. τₗ s = h • t \implies L ∧ k < t\}
8. \{∃t. τₗ s = h • t \implies L ∧ k < t\} ()

Fig. 8. Proof outline (and implementation) for lock. Here, tyLck binds the spec given to lock in Section 2.3.

(i.e., e : [x : A]. \{P x\} \{Q x\}), then x can be specialized by v : A, to derive e : \{P v\} \{Q v\}. The latter
follows because the logical variables are universally quantified in the definition of Hoare types (context Γ in (3)), and can thus be specialized just like any other universally quantified variable.¹⁴

3.6 Revisiting Spinlocks

To illustrate the inference rules, the proof outline in Figure 8 shows the proper implementation of
lock and the proof that lock has the type from Section 2.3 (the type is named tyLck in the figure).
The program is a loop executing CAS until it succeeds to lock. This is as in Section 2.3, except there
we informally bracketed CAS with the ghost code for manipulating histories, whereas here we
explicitly invoke the trylock_act action, which erases to CAS. The outline uses stable assertions
only: for example, the precondition in tyLck is stable, as argued in Section 2.3. Thus, we dispense
with explicit stabilization of assertions, i.e., applying (−)∗.

Given the Fix rule, in order to show that lock has the type tyLck, we must first prove that tyLck
is a loop invariant for fix, i.e., that it holds of the body of lock. Thus, the outline starts with the
precondition of tyLck in line 2, and derives the postcondition of tyLck in line 8. Line 3 derives
immediately from 2 and the definition of trylock_act in Section 3.2, equation (1), to expose that
trylock_act either succeeds to lock adding an L to the self history, of fails to lock keeping the
history unchanged.¹⁵ Notice that Line 3 has exactly the form required of a premise for the vrf_act
rule, with stabilization elided. Thus, the if–then–else conjunct in Line 3 is also a postcondition of
trylock_act, and therefore holds in line 5. Next we branch on b, which corresponds to applying the
rule vrf_cond. Line 6 considers the case b = true, and the postcondition immediately follows by the
rule vrf_ret (again, eliding stabilization). Line 7 considers the case b = false, and the postcondition
immediately follows as the recursive call to loop, by assumption, already has the desired type tyLck.
As both branches of the conditional have the same postcondition, the postcondition propagates to
line 8 to complete the proof.

4 EXCLUSIVE LOCKING VIA MORPHING AND THE NEED FOR PERMISSIONS

We next illustrate a more involved application of morphisms and simulation: how to derive a
resource and methods for exclusive locking, à la CSL [O’Hearn 2007], from the resource for spin

¹⁴We perform the described type changes silently in the paper. In Coq, they aren’t silent, but must be marked by a constructor.
Our implementation minimizes the number of such constructors, and makes them unobtrusive, but describing how is
beyond the scope of the paper.

¹⁵In the case of failure, we could also derive that the lock was taken at the moment trylock_act was attempted, i.e.
\exists t. τₗ s’ = h ∧ t s’ t = L ∧ k ≤ t ≤ last_stamp (τ s’). However, the rest of the proof doesn’t require the additional detail.
locks from Section 2. An exclusive lock protects a shared heap, satisfying a user-supplied predicate $R$ (aka. resource invariant). Upon successful locking, the shared heap is transferred to the private ownership of the locking thread, where it can be modified at will, potentially violating $R$. Before unlocking, the owning thread must re-establish $R$ in its private heap, after which, the part of the heap satisfying $R$ is moved back to the shared status. The idea is captured by the following methods and specs, which we name `exlock` and `exunlock` to differentiate from `lock` and `unlock` in Section 2.

\[
\begin{align*}
exlock & : \{ \alpha_s, \mu_s = \text{own} \land \chi_s = 0 \} \{ \lambda_s, \mu_s = \text{own} \land R(\chi_s s) \}@CSL \\
exunlock & : \{ \lambda_s, \mu_s = \text{own} \land R(\chi_s s) \} \{ \alpha_s, \mu_s = \text{own} \land \chi_s s = 0 \}@CSL
\end{align*}
\]

Here $\mu_s$ is a ghost of type $O = \{ \text{own}, \text{own} \}$, signifying whether "we" own the lock or not, and $\chi_s s$ is "our" private heap. $O$ has PCM structure with the join defined by $x \sqcup \text{own} = \text{own} \sqcup x = x$, so that $\text{own}$ is the unit of the operation. We leave own $\sqcup$ own undefined, to capture that the locking is exclusive, i.e., the lock can’t be owned by a thread and its environment simultaneously. Our goal in this section is to derive `exlock` and `exunlock` using morphing and simulations to "attach" to lock and unlock the functionality of transferring the protected heap between shared and private state.

The idea for doing so is pictorially shown in Figure 9, where the CSLX resource contains the state components and transitions describing the functionality needed for heap transfers. In particular, $\alpha_s s \in O$ keeps track of whether we own the lock or not, $\sigma_s$ and $\sigma_j$ are the private and shared heap respectively, and the transitions open_tr and close_tr move the heap from shared to private and back, respectively. We want to combine the Spin and CSLX resources as shown in the figure, by combining their state spaces, and coupling open_tr with lock_tr, and close_tr with unlock_tr, so that the transitions execute simultaneously. This will give us an intermediate resource CSLX', and a morphism $f : \text{Spin} \rightarrow \text{CSLX}'$ defined similarly to the morphisms in Section 2:

\[
\begin{align*}
f_\chi x s & \equiv s \setminus 1 \\
f_\Delta x \text{lock_tr} & \equiv \text{lock_tr} \bowtie \text{open_tr} \\
f_\Delta x \text{unlock_tr} & \equiv \text{unlock_tr} \bowtie \text{close_tr}
\end{align*}
\]
We will then restrict CSL’ into the CSL resource that we used in the specs for exlock and exunlock, as we shall describe. The components $\mu_s$ and $\chi_s$ used in these specs will be functions out of the state components of CSL.

However, if we try to carry out the above construction using the Spin resource from Section 2, we run into the following problem. Recall that Spin can execute unlock_tr whenever the lock is taken, irrespective of which thread took it. On the other hand, close_tr can execute only if “we” hold the lock. But, $f_\Delta s$ unlock_tr $= \text{unlock}_tr \Rightarrow \text{close}_tr$, and therefore, in states where others hold the lock, Spin may transition by unlock_tr, with CSL’ unable to follow by $f_\Delta$. Moreover, it’s impossible to avoid such situations by choosing a specific $f$-simulation $I$ that will allow unlock_tr to execute only if we hold the lock. Simply, there is no way to define such $I$ because we can’t differentiate in Spin between the notions of $\text{unlock}_tr$ being “enabled for us”, vs. “enabled for others, but not for us”, as unlock_tr is enabled whenever the lock is taken.

The analysis implies that we should have defined Spin in a more general way, as shown in Figure 9. In particular, Spin should contain the integer components $\pi_s / \pi_o$ which indicate if unlock_tr is “enabled for us” ($\pi_s > 0$), or not ($\pi_s = 0$), and dually for others. These will give us the distinction we seek, as we shall see. In line with related work, we call $\pi$ permission to unlock.\(^{16,17}\)

A thread may have more than one permission to unlock, which it can distribute among its children upon forking, who can then race to unlock. The addition of the new fields leads to the CSX of states where others hold the lock, and is needed in CSLX in order to describe the heap-transfer functionality independently of Spin. On the other hand, Spin keeps the locking histories in $\tau$. Thus, once Spin and CSLX are combined, the two components must satisfy

$$\omega(\tau_s s) = (\alpha_s s = \text{own})$$

$$\omega(\tau_o s) = (\alpha_o s = \text{own})$$

as a basic coherence property. Furthermore, we want to encode exclusive locking, so we must

\(^{10}\)In general, the design of resource’s permissions obviously and essentially influences how that resource composes with others. Some systems, such as CAP [Dinsdale-Young et al. 2010] and iCAP [Svendsen and Birkedal 2014], although they don’t consider morphisms and simulations, by default provide a permission for each transition of a resource. In our example, that would correspond to also having a permission for lock_tr. Full generality also requires external transitions that move permissions to and from an outside resource. In our Coq code, these are used in the readers-writers example, to support non-exclusive locking. For simplicity, we elide such generality here, and consider only the permission to unlock, which suffices to illustrate morphisms and simulations.

\(^{11}\)Similar concepts arise in other concurrency models as well. For example, a transition in a Petri net fires only if there are sufficient tokens—akin to permissions—in its input places. The tokens are consumed upon firing.

\(^{12}\)Following Section 3.1.5, we make the specs small wrt. $\pi_s s$, for simplicity. By framing, lock’ can be invoked when $\pi_s s \geq 0$, in which case it increments $\pi_s s$ by 1.
require that only the thread that holds the lock has the permission to unlock:

\[
\begin{align*}
\pi_s s &= (\text{if } \alpha_s s = \text{own then } 1 \text{ else } 0) \\
\pi_o s &= (\text{if } \alpha_o s = \text{own then } 1 \text{ else } 0)
\end{align*}
\] (5)

(thus, \( \pi_s s, \pi_o s \in \{0, 1\} \), and at most one of them is 1).

Most importantly, the events recorded in the histories of Spin should correspond to exclusive locking, and thus:

\[
\tau_s s \perp_\omega \tau_o s
\] (6)

where \( h \perp_\omega k \) is defined as

\[
(\omega h \rightarrow \text{last\_stamp } k < \text{last\_stamp } h) \land \\
(\omega k \rightarrow \text{last\_stamp } h < \text{last\_stamp } k) \land h \perp k
\]

to say that if \( h \) (resp. \( k \)) indicates that a thread holds the lock, then another thread couldn’t have proceeded to add logs to its own history \( k \) (resp. \( h \)), and unlock itself.\(^\text{19}\)

It is now easy to see that \( Inv = (4) \land (5) \land (6) \) is an invariant of CSL’. The critical point is that (6) is preserved by the transition \( t = \text{unlock\_tr} \leadsto \text{close\_tr} \). Indeed, if in state \( s \in Inv \) we transition by \( t \), it must be \( \pi_s s > 0 \) by \( t \)’s definition, and thus \( \pi_s s = 1 \), and \( \pi_o s = 0 \), by (5). Also, we add a fresh \( U \) entry to the ending state \( s’ \), thus making \( \text{last\_stamp} (\tau_s s’) > \text{last\_stamp} (\tau_o s’) \). For (6) to be preserved, it must then be \( \omega (\tau_o s’) = \omega (\tau_o s) = \text{false} \), i.e., the lock wasn’t held by another thread. But this is guaranteed by (4), (5) and \( \pi_o s = 0 \). In other words, by using the permissions to unlock, we have precisely achieved the distinction that our previous definition of Spin couldn’t make.

Because \( Inv \) is invariant, we can construct a resource CSL out of CSL’, where \( Inv \) is imposed as an additional property of the underlying PCM and state space of CSL’. Indeed, our theory ensures that the set \( \Sigma (CSL’) \cap Inv \) can be made a global predicate, and thus be used as a state space of a new resource CSL. By Definition 3.2, globality depends on the underlying PCM, hence the construction involves restricting the PCM of CSL’ by \( Inv \). The mathematical underpinnings of such restrictions involve developing the notions of sub-PCMs, PCM morphisms and compatibility relations, which we carry out in [Nanevski et al. 2019b, Appendix B]. Here, it suffices to say that the construction leads to the situation summarized by the following diagram:

\[
\text{Spin} \xrightarrow{f} \text{CSL}’ \xrightarrow{i} \text{CSL}
\]

where morphism \( i \) is defined by \( i_\Sigma : s = s \) and \( i_\Delta : t = t \). Intuitively, CSL states are a subset of CSL’ states satisfying \( Inv \), and \( i_\Sigma \) is the injection from \( \Sigma (CSL) \) to \( \Sigma (CSL’) \).

This gives us the CSL resource, but we still need to transform lock’/unlock’ into exlock/exunlock, respectively. We thus introduce the following property on CSL states:

\[
Sim s \equiv \text{if } \alpha_s s = \text{own then } R(\sigma_s s) \text{ else } \sigma_s s = \emptyset
\]

which says that the self heap satisfies the resource invariant \( R \) iff the thread owns the lock. \( Sim \), unlike \( Inv \), is not an invariant, because it is perfectly possible for a thread to own the lock, but for its heap to not satisfy \( R \), because the thread has modified the acquired heap after locking it. However, \( Sim \) is an \( (i \circ f) \)-simulation, as it satisfies the commuting diagrams from Figure 7. For example, when Spin executes lock\_tr, then CSL sets \( \alpha_s s = \text{own} \) and acquires the shared heap, thus making the self heap satisfy \( R \). When Spin executes unlock\_tr, then CSL returns the shared heap, making the self heap empty. In other words, \( Sim \) describes the state of CSL immediately after locking, and immediately before unlocking, which suffices for the morphing of lock’ and unlock’. We only show

\(^\text{19}\)Requirement (6) restricts only the last timestamp in \( h \) and \( k \), not all timestamps hereditarily. This suffices for our proof.
the derivation for \( \text{exunlock} = \text{morph} \ (i \circ f) \ \text{unlock} \), and refer to the Coq code [Nanevski et al. 2019a] for the derivation of exlock, which is similar.

1. \( \{\alpha_s \ s = \text{own} \land R(\sigma_s \ s)\} \)
2. \( \{\tau_s \ s = \hat{\text{h}} \land \tau_s \ s = 1 \land \alpha_s \ s = \text{own} \land R(\sigma_s \ s)\} \)
3. \( \{\tau_s \ s = \hat{\text{h}} \land k = \text{last\_stamp} (\hat{\tau} \ s) \land \pi_s \ s = 1 \land \text{Sim} \ s\} \)
4. \( \text{morph} \ (i \circ f) \ \text{unlock}' // \text{using simulation Sim} \)
5. \( \{(\tau_s \ s = \hat{\text{h}} \bullet t \implies U \land k < t \land \pi_s \ s = 0 \lor \\
\tau_s \ s = \hat{\text{h}} \land k \leq t \land \hat{\tau} \ s \ t = U \land \pi_s \ s = 1 \land \text{Sim} \ s\} \)
6. \( \{\tau_s \ s = \hat{\text{h}} \bullet t \implies U \land k < t \land \pi_s \ s = 0 \land \text{Sim} \ s\} \)
7. \( \{\alpha_s \ s = \text{own} \land \sigma_s \ s = \emptyset\} \)

The key step is in line 6, where we must derive that the second disjunct in line 5 is false; that is, no thread could have unlocked before us in line 4. We infer this by reasoning about the histories \( \tau_s \ s \) and \( \tau_o \ s \) in line 5. From \( \pi_s \ s = 1 \), it must be \( \alpha_s \ s = \text{own} \), and then \( \omega (\tau_s \ s) = \text{true} \), by the invariant \( \text{Inv} \) which holds throughout, as \( s \) is a CSL state. By \( \text{Inv} \) again, \( \tau_s \ s \perp_\omega \tau_o \ s \), so last\_stamp \( (\tau_o \ s) \) < last\_stamp \( (\tau_s \ s) \). Thus, it must be last\_stamp \( (\tau_s \ s) \) = last\_stamp \( (\hat{\tau} \ s) \) = \( k \), because last\_stamp \( (\hat{\tau} \ s) \) is the maximum of last\_stamp \( (\tau_s \ s) \) and last\_stamp \( (\tau_o \ s) \). But this contradicts that \( \hat{\tau} \ s \) contains entry \( U \) at \( t \geq k \). We can now derive line 7: \( \alpha_s \ s = \text{own} \) follows from \( \pi_s \ s = 0 \) and \( \text{Inv} \), and \( \sigma_s \ s = 0 \) follows by \( \text{Sim} \). Finally, we obtain the desired spec of exunlock from the beginning of the section, by letting \( \mu \) be \( \alpha \) and \( \chi \) be \( \sigma \).

5 QUIESCENCE AND Indexed Morphism Families

The previous examples were about extending Spin the functionality of another resource, Counter or CSLX. In this section, we apply resource morphism not to extend a resource, but to restrict it, specifically by “forgetting” its ghost state. This is a feature commonly required when installing one resource into a private state of another. We need a slight generalization, however, to indexed morphism families (or just families, for short), as follows.

A family \( f: V \xrightarrow{X} W \) introduces a type \( X \) of indices for \( f \). The state component \( f_\Sigma: X \rightarrow \Sigma (W) \rightarrow \Sigma (V) \) and the transition component \( f_\Delta: X \rightarrow \Delta (W) \rightarrow \Delta (V) \) now allow input \( X \), and satisfy a number of properties, listed in [Nanevski et al. 2019b, Appendix C], that reduce to Definition 3.9 when \( X \) is the unit type. Similarly, \( f \)-simulations must be indexed too, to be predicates over \( X \) and \( \Sigma (W) \), satisfying a number of properties which reduce to Definition 3.11 when \( X = \text{unit} \).

The morph constructor and its rule are generalized to receive the initial index \( x \), and postulate the existence of an ending index \( y \) in the postcondition, as follows:

\[
\text{morph} \ f \ x \ e : \{\lambda s_w. \ (f \ x)^P \ s_w \land I \ x \ s_w\} \ \{\lambda s_w. \ \exists y. (f \ y)^Q \ s_w \land I \ y \ s_w\} @ W \]

where \( (f \ x)^R \ s_w \cong \exists s_v. \ s_v = f_\Sigma \ x \ s_w \land R \ s_v \)

To illustrate, consider the resource Stack (Figure 10) implementing concurrent stacks, and the following spec for the stack’s push method, similar to that of lock from Section 2.

\[
\text{push}(v) : [k]. \ \{\lambda s. \ \sigma_s \ = \emptyset \land \tau_s \ = \emptyset \land k \leq \text{last\_stamp} (\tau_o \ s)\} \\
\{\lambda s. \ \sigma_s \ = \emptyset \land \exists t \ \text{vs}. \ \tau_s \ = t \implies (v s, v :: \text{vs}) \land k < t\} @ \text{Stack}
\]

Here \( \tau_s \) and \( \tau_o \) are histories of stack’s operations, as in the case of Spin, and \( \sigma_s \) is the thread-private heap. The spec says that push starts with \( \tau_s = \emptyset \) (by framing, any history) and ends with \( \tau_s = t \implies (v s, v :: \text{vs}) \) to indicate that a push of \( v \) indeed occurred, and after all the timestamps from the pre-state. The joint heap \( \sigma_f \) stores the stack’s physical layout, and \( \delta s \) is the abstract
The move from Stack to Priv thus essentially forgets the ghost state of histories, and the distinction in Stack between shared and private heaps. These components and distinctions are visible when in the scope of Stack, but hidden when in Priv. We would like to obtain the spec (7) by applying the \( \text{Mor}_H \) rule to the Stack spec of \( e \), with a morphism \( g : \text{Stack} \to \text{Priv} \) that forgets the histories. Unfortunately, such a morphism can’t be constructed as-is. Were it to exist, then \( g_E \), being contravariant, should map a state \( s_{\text{Priv}} \), containing only heaps, to a state \( s_{\text{Stack}} \), containing heaps and histories; thus \( g_E \) must “invent” the history component out of thin air.

\[ \{\lambda s. \text{layout } \text{nil}(\sigma_s,s)\}\{\lambda s. \text{layout } [a,b](\chi_s,s) \lor \text{layout } [b,a](\chi_s,s)\} @ \text{Priv} \]
This is where families come in. We make \( g : \text{Stack} \rightarrow \text{Priv} \) a family over \( X = \text{Hist} \), thereby passing to \( g_x \) the history \( x \) that should be added to \( s_{\text{Priv}} \) to produce an \( s_{\text{Stack}} \) (Figure 11(2)).

\[
g_x \times s_{\text{Priv}} = s_{\text{Stack}} \cong \chi_x (s_{\text{Priv}}) = s_{\text{Stack}} \cdot \sigma_j (s_{\text{Stack}}) \land \\
\chi_o (s_{\text{Priv}}) = s_{\text{Stack}} \land \\
\tau_x (s_{\text{Stack}}) = x \land \tau_o (s_{\text{Stack}}) = \emptyset
\]

The first conjunct directly states that Stack is installed in \( \chi_x (s_{\text{Priv}}) \) by making \( \chi_x (s_{\text{Priv}}) \) be the join of the heaps \( \sigma_j (s_{\text{Stack}}) \) and \( \sigma_x (s_{\text{Stack}}) \). The second conjunct says that the heap \( \chi_o (s_{\text{Priv}}) \) of the interfering threads is propagated to \( \sigma_o (s_{\text{Stack}}) \). The third conjunct captures that the history component of \( s_{\text{Stack}} \) is set to the index \( x \), as discussed immediately above. In the last conjunct, the \( \tau_o (s_{\text{Stack}}) \) history is declared \( \emptyset \), thus formalizing quiescence. We elide the definition of \( g_{\Lambda_1} \); it suffices to know that it maps a Stack transition (relation over heap and ghost state of Stack) to its “erasure”, i.e., a relation over heaps of Priv representing a single-pointer operation such as read, write and CAS, but ignoring the ghost histories.

We can now obtain the quiescent spec (7) by applying the MorphX rule to \( e \), as shown below, with \( x = \emptyset \), and \( I x \) being the always-true predicate on Priv states (the outline expands the definition of \( (g x) \)., and elides the always-true \( I x \).

1. \( \{ \text{layout nil} (\chi_x (s_{\text{Priv}})) \} \)
2. \( \{ \exists \text{Stack} \cdot s_{\text{Stack}} = g_x \emptyset s_{\text{Priv}} \land \sigma_x (s_{\text{Stack}}) = \emptyset \land \tau_x (s_{\text{Stack}}) = \emptyset \} \)
3. \( \text{morph } g \emptyset e \)
4. \( \{ \exists y s_{\text{Stack}} = g_x y s_{\text{Priv}} \land \sigma_x (s_{\text{Stack}}) = \emptyset \land \\
\exists t_1 \upsilon s_1, t_2 \upsilon s_2, \tau_x (s_{\text{Stack}}) = t_1 \Rightarrow (\upsilon s_1, a :: \upsilon s_1) \land t_2 \Rightarrow (\upsilon s_2, b :: \upsilon s_2) \} \)
5. \( \{ \text{layout} [a, b] (\chi_x (s_{\text{Priv}})) \land \text{layout} [b, a] (\chi_x (s_{\text{Priv}})) \} \)

Line 2 derives trivially from line 1, as the state \( s_{\text{Stack}} \) is uniquely determined; i.e., take \( \sigma_j (s_{\text{Stack}}) \) to be the heap storing the empty stack, and all the other components of \( s_{\text{Stack}} \) to be empty. We next derive the postcondition in line 5. From the properties of \( \Sigma (\text{Stack}) \), we know layout \( (\hat{\tau} (s_{\text{Stack}})) (\sigma_j (s_{\text{Stack}})) \). From \( s_{\text{Stack}} = g_x y s_{\text{Priv}} \) and other conjuncts in line 4, we know \( \chi_x (s_{\text{Priv}}) = \sigma_j (s_{\text{Stack}}) \), and \( \tau_x (s_{\text{Stack}}) = y \), and \( \tau_o (s_{\text{Stack}}) = \emptyset \). Thus, it is also \( \hat{\tau} (s_{\text{Stack}}) = y \Rightarrow (\upsilon s_1, a :: \upsilon s_1) \land t_2 \Rightarrow (\upsilon s_2, b :: \upsilon s_2) \), and:

\[
\text{layout} (\hat{\tau} (s_{\text{Stack}})) (\chi_x (s_{\text{Priv}}))
\]

From the properties of \( \Sigma (\text{Stack}) \), we also know that \( \hat{\tau} (s_{\text{Stack}}) \) has no timestamp gaps; thus \( \{ t_1, t_2 \} = \text{dom} (y) = \{1, 2\} \), i.e., \( t_1 \) and \( t_2 \) are the only, and consecutive, events in \( y \). But then \( \hat{\tau} (s_{\text{Stack}}) \) must be either \([a, b]\) or \([b, a]\), which, with (8), derives the postcondition.

---

21As we want to build \( s_{\text{Stack}} \) out of \( s_{\text{Priv}} \), we have to identify the part of \( \chi_x (s_{\text{Priv}}) \) which we want to assign to \( \sigma_j (s_{\text{Stack}}) \). This part has to be uniquely determined, else \( g_x \) won’t be a function. We ensure uniqueness by insisting that the predicate layout is precise – a property commonly required in separation logics.
6 RELATED WORK

Coalgebra morphisms and refinement mappings. Transition systems are mathematically representable as coalgebras [Jacobs 2016; Rutten 2000]; thus, resource morphisms are closely related to coalgebra morphisms, with differences arising from our application to concurrent separation logic and types. For example, for us, given \( f : V \rightarrow W \), \( f_C \) is contravariant, but in the coalgebraic case, \( f \) is covariant on states [Hasuo et al. 2009; Rutten 2000]. A coalgebra morphism \( f \) doesn’t have the \( f_\Delta \) component, but requires that \( f \) preserve and reflect the \( V \)-transitions (i.e., if \( x \rightarrow y \) is a \( V \)-transition, then \( f(x) \rightarrow f(y) \) is a \( W \)-transition, and if \( f(x) \rightarrow t \) is a \( W \)-transition, then there exists \( y \) such that \( x \rightarrow y \) is a \( V \)-transition, and \( f(y) = t \)). These properties are similar in spirit to the two clauses of our Definition 3.11 of \( f \)-simulations.

Similarly, refinement mappings [Abadi and Lamport 1991; Lynch and Vaandrager 1995], like coalgebra morphisms, are covariant functions on STSs, and differ from resource morphisms in the intended use. For example, with refinement mapping, the key question is how to extend the source STS with ghost state. The extension may be necessary, as a refinement mapping need not exist otherwise. In contrast, we seek to give a definitive resource type to a program, which suffices for all reasoning. The program can be morphed, to change the type, but can’t be re-typed by an extended resource, as that violates compositionality in our setting. Given a morphism \( f : V \rightarrow W \), the resource \( W \) will most commonly generalize and include \( V \)’s functionality. Thus, \( f_W \) can compute \( s_V \in \Sigma(V) \) out of \( s_W \in \Sigma(W) \), without needing to extend \( V \) or \( W \). When \( s_W \) lacks information to compute \( s_V \) (cf. the quiescence example in Section 5), we don’t extend the resources, but pass the missing information by an index in a morphism family. Morphisms also exhibit a form of (weak) simulation of \( W \) by \( V \) on the transposed states (Definition 3.11(2)).

We further establish the action of \( f \) on programs, and provide a Hoare logic rule to reason about it, supporting the usual compositionality notions from separation logic, such as framing and ownership transfer, which haven’t been considered in the context of coalgebra morphisms and refinement mappings.

Linearizability. In a relational flavor of separation logics [Frumin et al. 2018; Liang et al. 2012; Turon et al. 2013], and more generally, in the work on proving linearizability [Bouajjani et al. 2017; Gu et al. 2015, 2018; Henzinger et al. 2013; Khyzha et al. 2017; Liang and Feng 2013; Schellhorn et al. 2012], the goal is to explicitly relate two programs, typically one concurrent, the other sequential. The sequential program then serves as a spec for the concurrent one, and can replace it in any larger context. Our goal in this paper is somewhat different; we seek to identify the concurrent program’s type, which for us takes the form of a Hoare triple enriched with a resource. The type serves as the program’s interface, and, as standard in type theory, any two programs with the same type can be interchanged in clients’ code and proofs. As clients can already reason about the program via this type, the program shouldn’t need a spec in the form of another program.

There are several advantages of our approach over linearizability. First, a spec in the form of a Hoare triple with a resource is much simpler, and thus easier to establish than a spec in the form of another program. Indeed, resources aren’t programs; they are STSs and don’t admit programming constructs such as conditionals, loops, initial or local state, or function calls. A Hoare spec is also immediately useful in proofs, whereas with linearizability, one also has to verify the sequential program itself. Second, in linearizability it has traditionally been difficult to address ownership transfer of heaps between data structures [Cerone et al. 2014; Gotsman and Yang 2012], whereas for us (and other extensions of CSL [da Rocha Pinto et al. 2014; Dinsdale-Young et al. 2010; Jung et al. 2018, 2015; Liang and Feng 2013]), ownership transfer is directly inherited from separation logic. We also inherit from separation logic a way to dynamically nest parallel compositions of
threads, whereas linearizability is typically considered on programs with a fixed, though arbitrary, number of threads.

That said, we note that our specs of lock and unlock in Section 2 are actually very close to what one gets from linearizability, as they essentially establish a linear order between locking and unlocking events in the history PCM. In fact, our approach seems sufficiently powerful to directly specify, in the state space of a resource, the general property of linearizability as a user-level proposition over the resource’s subjective histories, which is the direction we intend to pursue in the future. Such a development would generalize the current paper, in that the events tracked by histories wouldn’t be instantaneous, as was the case in Section 2, but would have non-zero duration. The histories would have to record the events’ beginning as well as ending times.

Morphisms and simulations will play a key role in such a setting. For example, two different resources will have two different spaces of timestamps (i.e., two different clocks). Combining the two into a larger resource, will require constructing a history for the combination, of which the first step is reconciling the clocks of the two components into a common clock. Morphisms and simulations will be a necessary abstraction to relate the common clock, timestamps, and history to the clocks, timestamps, and histories of the components.

In future work, we thus expect to incorporate general linearizable programs and resources; note, however, that the PCM and history-based approach is further general still, and capable of compositionally specifying and verifying non-linearizable programs as well [Sergey et al. 2016].

State transition systems and abstract atomicity. Abstract atomicity refers to extending the functionality of a concurrent program so that it operates over other STSs. One of the most powerful approaches to abstract atomicity has been parametrization by auxiliary code. For example, in the case of spin locks, a way to make lock operate over an extension of Spin with an unknown STS X, is to parametrize lock a priori with an unknown ghost function α over X, thus modifying it into:

\[ \text{lock } \alpha \triangleq \text{do } \langle x \leftarrow \text{CAS}(r, \text{false}, \text{true}); \text{if } x \text{ then } \alpha \rangle \text{ while } \neg x \]

This differs from our approach, where the modification to lock is done a posteriori, and is mediated by the resource type of the program. Parametrization originated in Jacobs and Piessens [2011], and was extended to impredicative higher-order ghost functions and state in HOCAp [Svendsen et al. 2013], iCAP [Svendsen and Birkedal 2014] and Iris [Jung et al. 2018, 2015]). Parametrization affords abstract specifications that are similar to specifications that one would ascribe to a data structure in the sequential setting. It applies to the locking examples, as illustrated above, but also to many of the examples that we included in the Coq code. Parametrization relies on higher-order and impredicative ghost functions to specify a number of concurrency idioms, such as e.g., fork/join concurrency, or storing of concurrent programs into the heap. In contrast, our present model is predicative (see [Nanevski et al. 2019b, Appendix D]); hence we currently support the more restrictive concurrency by parallel composition.

Thus our approach doesn’t presently extend the range of verifiable programs. Rather, it proposes novel type-based foundations that underpin the verification, and that employ morphisms, histories, and simulations. Histories and simulations are foundational abstractions in concurrency. Morphisms are similarly so in mathematics, and in our case, relate to coalgebras and refinement mappings, as discussed. It’s therefore of inherent interest to embed these abstractions into type theory and obtain a minimalistic proof system for separation logic, as we have done.

By relying on these foundational abstractions, we achieve some uniformity and simplicity of reasoning, which we expect to build on in the future. For example, one challenge to parametrization arises when the point at which to execute the ghost function can be determined only after the program has terminated. This is a common pattern when proving linearizability, as exhibited, say, by the queue of Herlihy and Wing [1990], but has been difficult to address by parametrization,
because it isn’t clear at which point in the code to invoke the parametrizing ghost function. In our case, histories separate the termination of operations from their order in the linearization. The order becomes just another ghost component that can be constructed at run time, with cooperation of other threads [Delbianco et al. 2017]. A history-based spec can say that an operation finished executing, but that its exact place in the linearization is to be fully determined only later, by the action of other threads. We thus expect that history-based specs will support the described pattern of linearizability proofs.

The TaDA logic of da Rocha Pinto et al. [2014] introduces another approach to abstract atomicity that doesn’t rely on parametrization. TaDA defines a new judgment form, $\langle P \rangle e \langle Q \rangle$, that captures that $e$ has a precondition $P$ and postcondition $Q$, but is also abstractly atomic in the following sense: $e$ and its concurrent environment maintain the validity of $P$, until at one point $e$ takes an atomic step that makes $Q$ hold. Afterwards, $Q$ may be invalidated, either by future steps of $e$, or by the environment. Once judged atomic, programs can be associated with ghost code of other resources. In this sense, TaDA’s extension of code is a posteriori, similar to ours. In contrast to TaDA, we specify programs using ordinary Hoare triples, but rely on the PCM of histories to express atomicity: a program is atomic if it adds a single entry to the self history. In the Coq files, we have also applied morphisms to algorithms with helping, such as the flat combiner [Hendler et al. 2010], where one thread executes work on behalf of others; helping is an idiom that TaDA currently can’t express.

Sergey et al. [2018] have designed a logic Disel for distributed systems, in which one can combine distributed protocols—represented as STSs—by means of hooks. A hook on a transition $t$ prevents $t$ from execution, unless the condition $P$ associated with the hook is satisfied. In this sense, hooks implement an instance of our transition coupling, where one operand is fixed to the idle transition with a condition $P$, i.e. $\text{id}_{\text{tr}}P = \lambda s s' . P \land s \land s' = s$. Disel doesn’t currently consider hooks where both operands are non-idle, which we used in the lock examples, or notions of morphism and simulation. On the other hand, we haven’t considered distribution so far.

Finally, while morphisms and simulations provide a solution to abstract atomicity, they also go beyond it. For example, they apply to quiescence (Section 5), which, unlike abstract atomicity, doesn’t extend functions to resource combinations, but forgets the histories of a resource. They may also provide a foundation for answering more basic, categorical, questions about concurrent structures, such as, e.g., “when are two resources isomorphic” (answer: when there are cancelling morphisms between them). We plan to explore such questions in the future.

Automated separation logics for concurrency. A number of recent automated tools such as Verifast [Jacobs et al. 2011], VerCors [Amighi et al. 2018; Blom and Huisman 2014], and Viper [Müller et al. 2016], address the reasoning about concurrent programs in various extensions of separation logic. In general, the tools address fragments of Java or C, and completely or partially automate the discharge of the proof obligations. Our paper is accompanied with an implementation in Coq as a shallow embedding. Hence, Coq plays a dual role for us: it’s a framework for mechanizing proofs, but also a concurrent programming language. The high-level difference from the automated tools is, or course, that proofs are developed interactively. The scaling of the proving effort is achieved by the reuse of programs and proofs, enabled by the compositional nature of the underlying type theory. We haven’t explored automation yet, but the minimalistic nature of our setting suggests that the underlying abstractions will be useful for both interactive and automated reasoning.

FCSL. The current paper adds to FCSL [Nanevski et al. 2014] the novel notions of resource morphism, and significantly modifies the notion of resources. In FCSL, each concurrent resource is a finite map from labels (natural numbers) to sub-components. For example, using the concepts from Section 2, one could represent SC as a finite map $l_1 \Rightarrow \text{Spin} \uplus l_2 \Rightarrow \text{Counter}$, where $l_1$ and $l_2$ are
labels identifying Spin and Counter, respectively. This approach provides interesting equations on resources; for example, one can freely rearrange the finite map components by using commutativity and associativity of disjoint union $\cup$. However, it also complicates mechanized proofs, because one frequently, and tediously, needs to show that a label is in the domain of a map, before extracting the labeled component. In the current work, states aren’t maps, but triples which are combined by a form of pairing (e.g., the PCM of SC is a product of PCMs of Spin and Counter). Consequently, if we changed the definition of SC in Section 2 into SC’ by commuting Spin and Counter throughout the construction, then SC and SC’ wouldn’t be equal resources, but they will be isomorphic; in that we could exhibit cancelling morphisms between the two. But this requires first having a notion of morphism, which is one of the technical contributions of this paper. FCSL supported quiescence by means of a dedicated and very complex inference rule, whereas Section 5 demonstrates that quiescence is merely an application of morphism families.

7 CONCLUSIONS

This paper develops novel notions of resource morphisms and associated simulations, as key mathematical concepts that underpin a separation logic for fine-grained concurrency. This is a natural development, as structures in mathematics are always associated with an appropriate notion of morphism, and simulations are the invariants that the morphisms preserve. Morphisms and simulations act on programs, and are integrated into separation logic via a single inference rule that propagates the simulation from the precondition to the postcondition of the morphed program. Morphisms compose and can support different constructions and applications. One application is abstract atomicity, whereby a general spec, such as the one for lock in Section 2, is specialized to a specific ownership discipline, e.g., exclusive locking in Section 4. Other applications include the managing of scope of ghost state in quiescent environments, as illustrated in Section 5, and restricting the state space of a resource with additional state and PCM invariants, as used in Section 4.

ACKNOWLEDGMENTS

We thank Jesús Domínguez, Constantin Enea, František Farka, Joakim Öhman, Exequiel Rivas Gadda, Mihaela Sighireanu, Ana Sokolova, Gordon Stewart, Anton Trunov and Nikita Zyuzin for their comments on the various drafts of the paper. We thank the anonymous reviewers from OOPSLA’19 PC and AEC for their feedback. This research was partially supported by the Spanish MICINN projects BOSCO (PGC2018-102210-B-I00) and TRACES (TIN2015-67522-C3-3-R), the European Research Council projects Mathador (ERC2016-COG-724464) and FOVEDIS (ERC2015-STG-678177), and the US National Science Foundation (NSF). Any opinions, findings, and conclusions or recommendations expressed in the material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES


Afshin Amighi, Marieke Huisman, and Stefan Blom. 2018. Verification of shared-reading synchronisers. In *International Workshop on Methods and Tools for Rigorous System Design (MetTиD@ETAPS)*. 107–120. [https://doi.org/10.4204/eptcs.272.9](https://doi.org/10.4204/eptcs.272.9)


