Resource Morphisms for Specifying Concurrent Programs in Separation Logic

Aleksandar Nanevski∗, Anindya Banerjee∗, Germán Andrés Delbianco†, Ignacio Fábregas∗
∗IMDEA Software Institute, Madrid, Spain
†IRIF - Université Paris Diderot, Paris, France

Abstract—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. We apply morphisms to abstract atomicity, where a program verified under one resource is adapted to operate under another resource, thus facilitating proof reuse.

I. 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 prominent solutions to the problem, such as linearizability [1], more generally contextual refinement [2–4]. Most recently, Concurrent Separation Logic (CSL) [5, 6] and its many extensions to fine-grained concurrency, have approached the problem using compositional reasoning principles inherent in program logics, e.g. Hoare logic. An important aspect of these extensions of CSL is that method specs are enriched with additional algebraic notions in order to codify some important properties of fine-grained behavior of concurrent programs on their underlying data structures.

More specifically, state transition systems [7–12] termed resources [5, 13, 14] in this paper—describe concurrent data structures, as follows: a resource’s state space describes the structure’s invariants, and the resource’s transitions describe the permissible atomic changes to the structure’s state. Resources thus combine ideas from the resource invariant method of Owicki and Gries [14] and the rely-guarantee method [15], to facilitate modular reasoning in the presence of fine-grained interference. Some approaches [16, 17] employ resources to establish contextual refinement itself, suggesting that resources are mathematically foundational, and underpin other properties such as linearizability and refinement.

The key contributions of this paper are a novel notion of resource morphisms, i.e. functions that preserve resource structure, and a way to incorporate them into separation logic by means of a single inference rule that uniformly handles many different functionalities that morphisms can encode. The upshot is conceptually simple foundations for separation logics for fine-grained concurrency.

Resource morphisms arise as a natural extension of the existing work on concurrent separation logics [8, 9, 11, 12, 18–20], because a structure in mathematics typically is associated with an appropriate notion of a structure-preserving function. Examples include vector spaces and linear maps, groups and group homomorphisms, cpo’s and continuous functions, functors 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.

In particular, resource morphisms provide a novel approach to the problem of abstract atomicity [11] in separation logic, which can be explained as follows. Consider a spin lock r (a shared boolean pointer) and a program that locks it by setting it to true, looping if r is already set1:

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

The program isn’t atomic because it performs potentially many compare-and-set (CAS) commands in a loop. However, it can be viewed as abstractly atomic by clients, for whom it’s enough to observe the effect of lock acquisition (i.e., CAS succeeds) and to ignore all previous, unsuccessful, loop iterations. Verifying such clients typically requires associating lock with some ghost code α that operates over the client’s own state, and executes simultaneously with the lock’s effect, essentially to inform the client that locking took place. The problem of abstract atomicity is how to formally represent and reason about this simultaneity. The state of the art [8, 9, 12, 18] is to parametrize lock by α, a priori, thus modifying it into:

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

where ghost code is given in gray background, and brackets indicate atomic execution during which other threads can’t run2. This makes the verification compositional, as one can verify lock α out of the spec of α. We discuss such approaches further in Section VI.

In contrast, our approach employs morphisms to re-interpret the program a posteriori, as follows. Because resource is part of the program’s specification, we treat it as (part of) the program’s type. Execution of a program “inhabiting” a resource is a path through the resource’s graph. For example,

1We assume that \(\text{CAS}(r, a, b)\) atomically sets the pointer \(r\) to \(b\) (and returns true) if \(r\) contains \(a\), and leaves \(r\) unchanged (and returns false) otherwise.
2As CAS is already atomic, and α is ghost code, the granularity of the program, and its behavior on real (i.e., non-ghost) state, remain unchanged.
lock can be typed by the resource Spin from Figure 1(a), and it describes paths through Spin consisting of several idle transitions id_tr, corresponding to unsuccessful CAS's, followed by a locking transition lock_tr.

Now, given resources V and W, a resource morphism \( f : V \to W \) consists of two partial functions \( f_\Sigma : \Sigma_V \to \Sigma_W \) and \( f_\Delta : \Delta_V \to \Delta_W \). If \( e \) takes a state in \( V \) 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 the commuting diagram in Figure 1(b), the morphed program is in \( W \), 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 \( V \) (explaining 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 process determines a program in \( W \) that we denote morph \( f e \). Morph is a program constructor, and a form of function application of \( f \) to \( e \). As Section II illustrates, in the lock example, the resource V is Spin, and W is a combination of Spin and the client resource. The \( f_\Sigma \) component of the morphism projects a Spin state out of the combined state of Spin and the client, while \( f_\Delta \) maps the lock_tr transition of Spin to a transition that combines lock_tr and \( \alpha \).

To reason about morph \( f e \), we introduce the notion of morphism-specific simulations. An \( f \)-simulation is a predicate \( I \) over \( V \)-states satisfying, among other conditions listed in Section III, the key technical property that the diagram 1(b) commutes, if \( I s_w \). Simulations provide a way to reason about morph \( f e \) compositionally out of \( e \), with the following single inference rule, which is our second contribution:

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

\[
morph \ f \ e : \{ \lambda s_w. f \neg P s_w \land I s_w \} \{ \lambda s_w. f \neg Q s_w \land I s_w \} @W
\]

Here \( f \neg P s_w \) is defined as \( \exists s_v. s_v = f_\Sigma s_w \land P s_v \). The rule directly translates diagram 1(b) into program specs. The precondition assumes \( I s_w \) and the existence of \( s_v = f_\Sigma s_w \) such that \( P s_v \). By the fault avoidance property [21] of separation logic, a program \( e \) that is ascribed a Hoare triple isn’t stuck; thus there exists a transition \( t_w \) by which \( e \) steps from \( s_v \) into \( s'_v \). Because \( I \) is an \( f \)-simulation, the commutation of the diagram 1(b) implies the existence of \( t_w \) and \( s'_w \) such that \( s'_v = f_\Sigma s'_w \) and \( I s'_w \). Iterating this process for \( s'_w \), and the subsequent states, until \( e \) terminates in some final state \( s''_w \), derives the postcondition above. First, \( Q s''_w \) must hold as \( Q \) is \( e \)'s postcondition. Second, \( I \) being an \( f \)-simulation yields a commuting diagram which implies the existence of \( t''_w \) and \( s''_w \) 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\(^5\), but also decomposed and constructive manner. Classically, W simulates V if whenever V takes a transition, there exists a transition for W to take. Instead, for us, morphism is separate from the simulation, and computes the witness of this existential in the style of constructive logic and type theory. Classically, a simulation is a relation on the states of V and W. For us, it is a predicate on W-states alone, as a V-state, if any, that is related to a W-state by an \( f \)-simulation, can be computed by \( f_\Sigma \).

Morphisms go beyond abstract atomicity, as they also support generic constructions over resources, such as e.g., “tensoring” 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 resource constructions.

We formalized the development in Coq, using Coq’s predicates over states as assertions\(^4\), building on the previous work on Fine-grained Concurrent Separation Logic (FCSL) [10]. The sources are available as a supplement [22]\(^3\).

II. BACKGROUND AND OVERVIEW

We illustrate our specification idiom, resources, and resource morphisms, by fleshing out the example of spin locks, where programs lock by looping, trying to change r from false to true by a CAS, and unlock by writing false into r.

Specifications. To specify the methods for locking and unlocking, we build on the idea of linearizability [1], 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” [27]. Specifically, a program state s contains a ghost component that we project as \( \tau_s s \), and which keeps “our” history of lock operations. Dually, the projection \( \tau_o s \)

\(^3\)Our logic has nine rules, each addressing an orthogonal linguistic feature.

\(^4\)As apparent from the morph rule, we explicitly bind the state \( s_w \), in contrast to separation logics where this state is implicit. Ours is still a separation logic, as it admits fault avoidance, small footprints, and framing.

\(^5\)In addition to the examples from the paper, the sources include further benchmarks such as Treiber stack [23], flat combiner [34], concurrent allocator, spanning tree of a graph, ticket [25] and readers-writers [26] locks.
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 $\tau_s$ and $\tau_o$ as self and other histories, respectively [10, 28], and denote by $\hat{\tau}$ the union history $\tau_s \ast \tau_o$.

$$\text{lock} : \{h, k\}, \{\lambda s. \tau_s = h \land k \leq \text{last\_stamp} (\hat{\tau})\}$$

$$\{\lambda s. \exists t. \tau_s = h \land t \Rightarrow L \land k < t\} \oplus \text{Spin}$$

$$\text{unlock} : \{h, k\}, \{\lambda s. \tau_s = h \land k \leq \text{last\_stamp} (\hat{\tau})\}$$

$$\{\lambda s. \exists t. \tau_s = h \land t \Rightarrow \bar{U} \land k < t \land \tau_o = h \land \hat{\tau} \land t = U \land k \leq t\} \oplus \text{Spin}$$

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, if the history $\tau_s$ equals $2 \Rightarrow U \bullet 7 \Rightarrow L \bullet 9 \Rightarrow L$, that 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 locked at time 8. The entries such as $2 \Rightarrow U$ are singleton maps, and $\bullet$ is disjoint union, undefined if operand histories share a timestamp.

Consider now the specs above. Lock starts with self history $\tau_s$ equal to $h^k$ in the precondition, 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 ($k \leq \text{last\_stamp} (\hat{\tau})$ in the precondition). The natural numbers ordering on timestamps gives the linear sequence in which the events logged in $\hat{\tau}$ occurred. Notice that the spec is stable, i.e., invariant under interference. Intuitively, other threads can’t modify the $\tau_s$ field, as it’s private to “us”. They can log new events into $\tau_o$, features which in the comparison $k \leq \text{last\_stamp} (\hat{\tau})$, 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 $t$ ($\hat{\tau} \ast t \Rightarrow U$). The conjunct $k < t$ captures that another thread may have freed $r$ after the invocation of unlock ($k < t$), or that we invoked unlock with $r$ already freed ($k = t$).

We emphasize that the specs for lock and unlock don’t require that the unlocking thread must be the one that locked (or even that the lock is taken when unlocking is attempted). We intend these specs to capture only the basic mechanics of spin locks, and leave it to the clients to supply application-specific policies. For example, in Section IV, we will show how to apply morphisms to lock and unlock, to derive exclusive locking as in CSL [5], where the locking thread acquires exclusive read-write access to the protected state, which it relinquishes by unlocking. Elided in this paper, but also possible [22, 29], is to morph lock and unlock into non-exclusive locking protocols, such as readers-writers locks [26, 30], where

\[\text{lock} = \{x \leftarrow \text{CAS}(r, \text{false}, \text{true})\};
\]

\[\text{unlock} = \{x \leftarrow \text{id}\};
\]

Note how the bracketed code in lock (and similarly, in unlock) essentially makes a choice, depending on the contents of $r$, between executing lock_tr or the idle transition. The former corresponds to CAS successfully setting $r$, the latter to CAS failing. Thus, we will abstractly view atomic operations as a choice between one or more transitions of the corresponding resource, rather than as bracketing of ghost with real code.

Morphisms. Consider next how to express a client of lock that, simultaneously with a successful lock, adds $n$ to the ghost component $\kappa_s$ of resource Counter (right half of Figure 2).

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

\[\text{lock} = \{x \leftarrow \text{CAS}(r, \text{false}, \text{true})\};
\]

\[\text{unlock} = \{x \leftarrow \text{id}\};
\]

By default, from now on, we elide the idle transition id_tr from diagrams.
We desire something like $\langle \text{lock}; \kappa_s s := \kappa_s s + n \rangle$ but this isn’t quite right as the atomic brackets would prevent other programs from running during the iterations of lock’s loop. To model that addition occurs simultaneously with the successful CAS of the last iteration in lock, we use morphisms as follows.

First, we “tensor” the resources Spin and Counter, as graphically indicated\(^8\) in Figure 2; that is, we create a new resource SC whose state is a pair of Spin and Counter states, and transitions are $\text{lock}_{\text{tr}} \bowtie \text{incr}_{\text{tr}} n$ and $\text{unlock}_{\text{tr}} \bowtie \text{id}_{\text{tr}}$. Operator $\bowtie$ (pronounced “couple”) indicates that the operand transitions are executed simultaneously on their respective state halves. It’s defined as follows, where $s\uparrow 1$ and $s\uparrow 2$ project state $s$ to its Spin and Counter components, respectively:

$$(t_1 \bowtie t_2) s s' \equiv t_1 (s\uparrow 1) (s\uparrow 1) \land t_2 (s\uparrow 2) (s\uparrow 2)$$

Second, we define a set of morphisms $f$ from Spin to SC, indexed by $n$, as follows:

$$\begin{align*}
(f n)^\Sigma s &= s\uparrow 1 \\
(f n)^\Delta s \text{ lock}_{\text{tr}} &= \text{lock}_{\text{tr}} \bowtie \text{incr}_{\text{tr}} n \\
(f n)^\Delta s \text{ unlock}_{\text{tr}} &= \text{unlock}_{\text{tr}} \bowtie \text{id}_{\text{tr}}
\end{align*}$$

This definition captures that (1) starting from an SC state $s$, we can obtain a Spin state by taking the first projection; and (2) a Spin program can be lifted to SC by changing their transitions by $(f n)^\Delta$ on-the-fly. We can now achieve the desired *a posteriori* re-interpretation of lock as

$$\text{morph } (f n) \text{ lock}$$

to obtain a program in the SC resource, which executes $\text{lock}_{\text{tr}} \bowtie \text{incr}_{\text{tr}} n$ whenever lock executes lock$_{\text{tr}}$, thus incrementing $\kappa_s s$ precisely, and only, upon a successful CAS.

*Inference.* The morph rule provides a way to reason about morphed programs. For example, the program

$$\begin{align*}
\text{morph } (f 1) \text{ lock}; \\
\text{morph } (f 42) \text{ unlock}; \\
\text{morph } (f 2) \text{ lock}
\end{align*}$$

annotes the invocations of the Spin commands by different morphisms, $f 1$, $f 42$ and $f 2$, to illustrate that programs can be morphed in a variety of ways, which correspond to annotating them with different ghost code. The following is a proof outline that the program above increments $\kappa_s s$ by 3.

1. $\{\kappa_s (s\uparrow 2) = n\}$
2. $\{\kappa_s (s\uparrow 2) = n \land \tau_s (s\uparrow 1) = h\}$
3. $\text{morph } (f 1) \text{ lock};$
   // $I_1 s \equiv \kappa_s (s\uparrow 2) = n + 2L (\tau_s (s\uparrow 1)) - 2L h$
4. $\{\kappa_s (s\uparrow 2) = n + 2L (\tau_s (s\uparrow 1)) - 2L h \land \tau_s (s\uparrow 1) = h \leftrightarrow t \Rightarrow L\}$
5. $\{\kappa_s (s\uparrow 2) = n + 1 \land \tau_s (s\uparrow 1) = h'\}$
6. $\text{morph } (f 42) \text{ unlock};$ // $I_{42} s \equiv \kappa_s (s\uparrow 2) = n + 1$
7. $\{\kappa_s (s\uparrow 2) = n + 1 \land (\tau_s (s\uparrow 1) = h' \lor \tau_s (s\uparrow 1) = h' \leftrightarrow t' \Rightarrow U)\}$
8. $\{\kappa_s (s\uparrow 2) = n + 1 \land \tau_s (s\uparrow 1) = h''\}$
9. $\text{morph } (f 2) \text{ lock}$
   // $I_2 s \equiv \kappa_s (s\uparrow 2) = n + 1 + 2L (\tau_s (s\uparrow 1)) - 2L h''$
10. $\{\kappa_s (s\uparrow 2) = n + 1 + 2L (\tau_s (s\uparrow 1)) - 2L h'' \land \\
\tau_s (s\uparrow 1) = h'' \leftrightarrow t'' \Rightarrow L\}$
11. $\{\kappa_s (s\uparrow 2) = n + 3\}$

We start with the precondition $\kappa_s (s\uparrow 2) = n$, where $n$ snapshots “our” current count. Line 2 uses $h$ to snapshot “our” history. In each application of morph, we apply the rule from Section I, with a simulation as indicated. The simulation holds of the pre-state, and remains invariant throughout the execution of the morphed program, and into the post-state. The proof can use different simulations for different instances of morph. For example, in lines 2-4, under the morphism $f 1$, the simulation $I_1$ states that the counter $\kappa_s s$ increments by the number of fresh L-entries in the history $(\tau_s h$ is the number of L-entries in $h)$. For the morphism $f 2$, $I_2$ says that $\kappa_s s$ increments by twice that number, and $I_{42}$ says that the counter doesn’t change, as $f$ associates counter increment with locking only.

We formally define simulations in Section III, but intuitively, each of $I_1$, $I_2$ and $I_{42}$ is a simulation for its respective morphism, because it satisfies the diagram (b) independently of the Spin transition $t_v$ chosen by the morphed program. For example, $I_1$ is preserved by $t_v = \text{lock}_{\text{tr}}$, because $\text{lock}_{\text{tr}}$ adds an L entry to $\tau_s$, and $(f 1)^\Delta s \text{ lock}_{\text{tr}} = \text{lock}_{\text{tr}} \bowtie \text{incr}_{\text{tr}} n$ increments $\kappa_s s$ by 1. But, $I_2$ is also preserved by $\text{unlock}_{\text{tr}}$, as $\text{unlock}_{\text{tr}}$ doesn’t add L entries, and $(f 1)^\Delta s \text{ unlock}_{\text{tr}} = \text{unlock}_{\text{tr}} \bowtie \text{id}_{\text{tr}}$ doesn’t change $\kappa_s s$.

It’s important that being a simulation is a property independent of the morphed program, as then the proofs can be derived compositionally, i.e. from the program spec, not code. Specifically above, each of $I_1$, $I_2$ and $I_{42}$ enables computing the end-value of $\kappa_s s$ from the end-value of $\tau_s s$, and the latter is described in the specs of lock and unlock. Thus, the code of lock and unlock isn’t required by the proof outline.

In the rest of the paper, we proceed to develop the concepts of resources, morphisms and simulations formally, and illustrate on additional examples and applications.

### III. Definitions of the formal structures

#### A. States

As apparent in Figure 2, different resources may contain different state components, e.g., $\tau$ of Spin and $\kappa$ of Counter. To enable the user to declare her own components, we parametrize resource states by two types. The type $M$ circumscribes the *self* and *other* components, and the type $T$ classifies the joint (aka., *shared*) state. Thus, a state $s = (a_s, a_j, a_o)$ is an $(M, T)$-state iff $a_s, a_o \in M, a_j \in T$. In the case of Spin, $M$ is histories, while for Counter, it is $\mathbb{N}$. Neither Spin nor Counter requires shared ghost state, so $T$ is the unit type for both\(^8\). We use $a_s (s)$, $a_j (s)$ and $a_o (s)$ as generic projections out of $s$, but introduce labels for readability; e.g., $\tau_s / \tau_o$ and $\kappa_s / \kappa_o$ for $a_s / a_o$ in Spin and Counter, respectively.

The *self* and *other* components of different threads aren’t independent, but offer thread-local views of the thread configuration. For example, imagine two threads $\theta_1$ and $\theta_2$ running in parallel. If the state of $\theta_1$ is $s = (a_s, a_j, a_o)$, then the state of $\theta_2$ must be the *transposed* $s^T = (a_o, a_j, a_s)$. Both states represent the same configuration, with $s$ being the view of $\theta_1$.

\(^8\)Section IV will show an example with a non-trivial $T$. 

\(^\text{8}\)The formal definition of tensoring is presented in Appendix A.
and $s^\tau$ that of $\theta_2$. The transposition captures that $\theta_1$ is the environment of $\theta_2$ and vice versa.

The specs must often combine components, cf. how histories were unioned by $\bullet$ to express freshness of timestamps in the specs of lock and unlock. To provide for the combination, $M$ is endowed with the structure of a partial commutative monoid (PCM), which is a triple $(M, \bullet, 1)$ where $\bullet$ (join) is a partial, commutative, associative, binary operation on $M$, with $1$ as the unit. Histories under disjoint union, nats under addition, and heaps under disjoint union, are all PCMs. Given $x, y \in M$, we write $x \perp y$ to mean that $x \neq y$ is defined.

Imagine three threads $\theta_1$, $\theta_2$ and $\theta_3$ running in parallel. Their respective states must have the forms $s_1 = (a_1, a_2 \bullet a_3)$, $s_2 = (a_2, a_3 \bullet a_1)$ and $s_3 = (a_3, a_1 \bullet a_2)$, as any two threads combined are the environment for the third. If $\theta$ is the parent thread of $\theta_1$ and $\theta_2$, then its state is $s = (a_1 \bullet a_2, a_3)$, since $\theta$ is the combination of $\theta_1$ and $\theta_2$, and has $\theta_3$ as the environment. We abbreviate as $s = s_1 \ast s_2$ the relationship between the parent state $s$ and the children states $s_1$ and $s_2$.

As apparent from the discussion, we will frequently move values back-and-forth between $a_1$ and $a_2$. To facilitate such surgery, we introduce the following operations on state, and notions of invariance under a change of viewing thread.

**Definition 1.** Let $p \in M$ and $s = (a_1, a_2, a_3)$ be an $(M, T)$-state. Then $s < p = (a_1 \bullet p, a_2, a_3)$ and $s > p = (a_1, a_2 \bullet a_3, p)$ are self- and other-framing of $s$ with the frame $p$, respectively.

**Definition 2.** A predicate $P$ on $(M, T)$-states is global if $P(s \triangleright p) \iff P(s < p)$ for every $(M, T)$-state $s$ and $p \in M$ such that $a_1(s) \perp p$ and $a_2(s) \perp a_3(s)$. A (partial) function $f$ on states is global if $f(s \triangleright p) = f(s < p)$ under the same conditions. Given a global $P$, the predicate $Q$ is $P$-general, if $P(s < p) \rightarrow Q(s \triangleright p)$, and similarly for functions.

Examples of global predicates from Section II, are $P(s) = \tau_s \perp \tau_s$, and $Q(s) = \Delta(s)$ used in Figure 2 to characterize Spin histories. $P$ and $Q$ are both defined in terms of $\tau_s = \tau_s \perp \tau_s$, $Q$ directly so, and $P$ because $\tau_s \perp \tau_s$ if $\tau_s \perp \tau_s$ 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. Therefore, both are invariant under framings, as these shuffle timestamps between $\tau_s$ and $\tau_0$, but don’t alter the total. In fact, $\tau$ itself is global.

Global predicates thus describe properties that are invariant to all threads over a data structure, which is why we next use them to represent state spaces of a resource.

**B. Resources**

**Definition 3.** An $(M, T)$ state space is a pair $\Sigma = (P, \Sigma)$, where $P$ is a global predicate on $(M, T)$-state and $\Sigma$ is a partial $P$-global function into heaps, called erasure, such that for every state $s$, $P(s)$ implies that $a_1(s) \perp a_0(s)$ and $r^\Sigma$ is defined. We write $s \in \Sigma$ to mean $P(s)$.

For example, the state space of Spin in Figure 2 includes the the property that $r$ isn’t a null pointer, to ensure that Spin’s erasure is defined (as heaps can’t contain the null pointer).

We can now lift the relation $s = s_1 \ast s_2$ to predicates, and define the separation logic conjunction $P \ast Q$, which will be used in the inference rules in Section III-E.

$$(P \ast Q)(s) \equiv \exists s_1, s_2. s = s_1 \ast s_2 \land P(s_1) \land Q(s_2)$$

**Definition 4.** A transition $t$ over $(M, T)$ state space $\Sigma$ is a binary relation on $\Sigma$ states, such that:

1. (partial function) if $t \downarrow s_1'$ and $t \downarrow s_2'$ then $s_1' = s_2'$.
2. (other-fixity) if $t \downarrow s'$, then $a_0(s) = a_0(s')$
3. (transition locality) if $t(s \triangleright p)$, then there exists $s'$ such that $x = s' \triangleright p$ and $t(s \triangleright p)(s' < p)$

A state $s$ is safe for $t$, if there exists $s'$ such that $t \downarrow s'$.

Property (1) is obvious. Property (2) captures that a transition can’t change the other-component, as it is private to other threads. However, a transition can read this component, c.f. how lock_tr in Figure 2 uses $\tau_s$ to compute a fresh timestamp. To explain transition locality (3), 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 $s < p$. If $\theta_1$ performs a transition $t(s \triangleright p)$, then by (3), the move can be seen as a transition of $\theta$ in the state $s < p$. Intuitively, the parent is “held responsible” for the transitions of its children.

Transition locality requires that the frame $p$ must be available in $a_3$, from which it is removed to be framed onto $a_2$. For example, we can’t frame an arbitrary history $p$ onto $\tau_s$ in the transitions of Spin in Figure 2, but only a history $p$ that can be taken out of $\tau_0$. In particular, a transfer of $p$ preserves the value of global functions, such as, e.g. $\tau$. Transition locality thus bears on how we frame specs that contain global functions and properties. For example, we can give lock (Section II) the following spec, which is small [21] wrt. the history $\tau_s$,

$$\text{lock} : [k]. \{ \lambda s. \tau_s = \emptyset \land k \leq \text{last}_\Sigma(\tau_s) \}$$

and then we frame the history $h$ onto $\tau_s$ to obtain the equivalent spec we actually presented:

$$\text{lock} : [h, k]. \{ \lambda s. \tau_s = h \land k \leq \text{last}_\Sigma(\tau_s) \}$$

Notice how the unframed version applies last_stamp to $\tau_0$, whereas the framed version uses last_stamp to $\tau_s$, to compensate for the fact that framing added $h$ to $\tau_s$ and thus removed it from $\tau_0$. Of course, if a spec doesn’t use global properties, but is stated solely in terms of self components (as is the case in classical separation logic where specs describe private heaps), then the effect isn’t evidenced. Thus, the component $\tau_0$ gives us a local way to work with global properties.

**Definition 5.** A $\Sigma$-transition $t$ is footprint preserving if $t(s \triangleright s')$ implies that $r^s \ast r^{s'}$ 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 2, 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 [22, 29]. External transitions encode transfer of data in and out of a resource, of which allocation and deallocation are an instance. When a resource requires allocation or deallocation, it can be tolerated with an allocator resource to exchange pointers through ownership transfer [10, 31] via external transitions. We elide further discussion, but refer to the Coq files for the implementation.

Definition 6. A resource is a tuple V = (M, T, Σ, Δ), where Σ is an (M, T, Δ) state space, and Δ a set of footprint preserving Σ transitions. We refer to V’s components as projections, e.g. Σ(V) for the state space, Δ(V) for the transitions, M(V) for the PCM, etc. A state s is V-state iff s ∈ Σ(V).

Definition 7. Given a resource V and V-states s, s’, we say that s other-steps by V to s’ (written s →* s’), if there exists a transition t ∈ Δ(V) such that t s’ = s t. The stabilization of a predicate P is P* ≝ ∀s’ s →* s’ → P(s’), where →* is the reflexive-transitive closure of →, P is stable if P = P*.

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 8. An action of type A in a resource V is a partial function a : Σ(V) −→ Δ(V) × A, mapping input state to output transition and value, which is local, i.e., if a (s ⊢ p) = (t, v) then a (s ⊢ p) = (t, v). The effect of a is the partial function [a] : Σ(V) −→ Σ(V) × A mapping input state to output state and value, defined as [a] s = (s’, v) iff ∃t a.s = (t, v) ∧ t s s’. 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 II, as the following action of type bool:

\[
\text{trylock\_act } s = \begin{cases} 
   (\text{lock\_tr}, \text{true}) & \text{if } \neg \omega(\hat{r}s) \\
   (\text{id\_tr}, \text{false}) & \text{otherwise}
\end{cases}
\]

The action is local, as it depends only on \(\hat{r}s\), which is invariant under framing. Similarly to transition locality, action locality ensures that an action performed by a child thread behaves the same when viewed by the parent.

We say that a erases to an atomic read-modify-write (RMW) command c [32], if [a] behaves like c when the states are erased to heaps. In other words, if [a] s = (s’, v), then \(c r s \triangleq (\hat{r}s', v)\). One may check that trylock\_act erases to CAS(r, false, true), as expected. Similarly, unlock\_act s = \[
\begin{cases} 
   (\text{unlock\_tr}, \lambda) & \text{if } \omega(\hat{r}s) \\
   (\text{id\_tr}, \lambda) & \text{otherwise}
\end{cases}
\]

\(^{10}\)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 the Coq implementation doesn’t currently issue proof obligations to check this.

is an action of unit type, which erases to \(r \leftarrow \text{false}\).

As actions return transitions, we are justified in viewing an execution of a program over resource V as a path through the graph that is V, as mentioned in Section I.

C. Morphisms

Definition 9. A morphism \(f : V \rightarrow W\) consists of partial functions \(f_\Sigma : \Sigma(W) \rightarrow \Sigma(V)\) (note the contravariance), and \(f_\Delta : \Sigma(W) \rightarrow \Delta(V) \rightarrow \Delta(W)\), such that:

1. (locality of \(f_\Sigma\)) there exists a function \(\phi : M(W) \rightarrow M(V)\) such that if \(f_\Sigma(s_w \triangleright p) = s_w\), then \(\exists s'_w\) such that \(s_v = s'_w \triangleright \phi(p)\), and \(f_\Sigma(s_w \triangleleft p) = s'_w \triangleleft \phi(p)\).

2. (locality of \(f_\Delta\)) if \(f_\Delta(s_w \triangleright p)(t_v) = t_w\), then \(f_\Delta(s_w \triangleleft p)(t_v) = t_w\).

3. (other-finity) if \(a_o(s_w) = a_o(s'_w)\) and \(f_\Sigma(s_w), f_\Sigma(s'_w)\) exist, then \(a_o(f_\Sigma(s_w)) = a_o(f_\Sigma(s'_w))\).

Morphism \(f\) transforms a V-program \(e\) into a W-program, as follows. When morph \(f\) is in a V-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 \(f_\Delta(s_w)(t_v)\).

The properties (1) and (2) of Definition 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 \(f_\Delta\) produces; thus it doesn’t influence the behavior of morphed programs. The property (3) restricts the choice of \(s'_w\) in (1) so that \(a_o(s'_w)\) 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 morphism family \(f_n : \text{Spin} \rightarrow \text{SC}\) from Section II. Indeed, \(M(\text{SC}) = M(\text{Spin}) \times M(\text{Counter}) = \text{histories} \times \mathbb{N}\). Thus, a frame in \(\text{SC}\) is a pair of a history and a nat; it is transformed into a frame in 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 \((f_n)\Delta\) 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 \(a_o(s'_w)\) are also pairs of a history and a nat. If the pairs are equal, then their history components are equal too, which derives (3).

D. Simulations

Because \(f_\Sigma\) and \(f_\Delta\) 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 \(s_v = f_\Sigma(s_w)\), whose computation is the first step of morphing, needn’t exist. Even if \(s_v\) does exist, and the original program takes the transition \(t_v\) in \(s_v\), then \(t_w = f_\Delta s_w t_v\) 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 typechecked against the morph rule doesn’t get stuck.

**Definition 10.** 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 \( fSensor(s_w) = s_w \), and \( t_v s_v s'_v \), then there exist \( t'_w = fDelta s_w t_v \) and \( s'_w \) such that \( I s'_w \) and \( fSensor(s'_w) = s'_v \), and \( t_w s_w s'_w \).
2. If \( I s_w \) and \( fSensor(s_w) \) exists, and \( s_w \xrightarrow{\Delta} s'_w \), then \( I s'_w \), and \( fSensor(s'_w) \) exists, and \( fSensor(s'_w) \xrightarrow{\Delta} fSensor(s'_w) \).

Property (1) states the \( W \) simulates \( V \) on states satisfying \( I \). Property (2) states the simulation in the opposite direction, i.e., of \( W \) by \( V \).

Finally, an **invariant** of a resource \( V \) (or \( V \)-invariant), is a predicate \( I \) which is preserved by all the transitions of \( V \). In other words, \( I \) is an \( id \)-simulation on \( V \).

**E. 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\(^{11}\).

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 (if \( A = \text{unit} \), we elide it from the type, as we did in Section II). The second 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.

The key concept in the inference rules is the predicate transformer \( vrf \ e \ Q \), which takes a program \( e : ST \ V \ A \) and postcondition \( Q \), and returns the set of \( V \)-states from which \( e \) is safe to run and produces a result \( v \) and ending state \( s' \) such that \( Q \ v \ s' \). Hoare triple are then defined in terms of \( vrf \).

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

We formulate the system using both \( vrf \) and the Hoare triples. The former is used, 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 \( T = \Pi_{e:A}. [\Gamma]. \ \{P\} \ B \{Q\}@V \):

\[
\text{fix} : (T \rightarrow T) \rightarrow T
\]

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 triples and typing for \( \text{fix} \), are the only Hoare-related rules of the system. In the rules we assume that \( e : ST \ V \ A, e_1 : ST \ V \ A_1, a \) is a \( V \)-action, \( f : V \rightarrow W \) is a morphism, \( I \) is an \( f \)-simulation, \( J \) is a \( V \)-invariant, and \( s \in \Sigma(V) \).

\[
\begin{align*}
\text{vrf\_post} & : (\forall v. J s \rightarrow Q_1 v s \rightarrow Q_2 v s) \rightarrow J s \rightarrow vrf \ e \ Q_1 s \rightarrow vrf \ e \ Q_2 s \\
\text{vrf\_ret} & : (Q v)^* s \rightarrow vrf (\text{ret} \ v) \ Q s \\
\text{vrf\_bind} & : vrf e_1 (\lambda x. vrf (e_2 x) \ Q) s \rightarrow vrf (x \leftarrow e_1; (e_2 x)) \ Q s \\
\text{vrf\_par} & : (vrf e_1 \ Q_1) \ast (vrf e_2 \ Q_2) s \rightarrow vrf (e_1 || e_2) (\lambda x. a_1 \times_2 (\lambda v. a_1 \times_2 (Q_1 v, 1)) \ast (Q_2, 2)) s \\
\text{vrf\_frame} & : (vrf e_1 \ Q_1) \ast \text{Q}^* \ s \rightarrow vrf e_2 (\lambda v. a_1 \times_2 (\lambda v. a_1 \times_2 (Q_1 v) \ Q_2) s \\
\text{vrf\_act} & : (\lambda s'. \exists s'' v. [a] s'' = (s'', v) \land (Q v)^* s'') s \rightarrow vrf (a) \ Q s \\
\text{vrf\_morph} & : \text{f}'(vrf e \ Q) s \rightarrow I s \rightarrow vrf (\text{morph} \ e \ Q) (\lambda v. s', f'(Q v) s'/I s'') s \\
& \text{where } (f \ P)^* \equiv fSensor (e) \text{ exists and } P (fSensor (e))
\end{align*}
\]

The vrf\_morph rule is a straightforward casting of the morph rule from Section I into a predicate transformers style. Similarly, vrf\_par and vrf\_frame are predicate transformer variants of the rules for parallel composition and framing in separation logic. Whereas the frame rule of separation logic assumes that the frame predicate \( Q_2 \) is stable, here we stabilize it explicitly by using \( Q_2^* \). vrf\_bind is a Dijkstra-style rule for sequential composition, and vrf\_ret is the rule for the idle program, returning \( v \). It explicitly stabilizes \( Q \) to allow for the state \( s \) to be changed by interference after returning. vrf\_post weakens the postcondition, except that in doing so, it allows assuming a \( V \)-invariant \( J \). Thus, resource invariants can be elided from
\footnote{Appendix C 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.}
Hoare triples, as they can be invoked by vrﬁ_post as needed. Finally, vrﬁ_act says that Q holds after executing action a in state s, if s obtains s’ by interference of other threads, and then [a] s’ returns the pair (s”, u) of output state and value. The latter satisfy the stabilization of Q, to allow for interference on s” after the termination of a.

IV. 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 [5], from a resource for general locking (Section II). 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 II.

\[
\text{exlock : } \{ \lambda s. \mu_s s = \text{ow} \wedge \chi_s s = \emptyset \} \\
\text{exunlock : } \{ \lambda s. \mu_s s = \text{ow} \wedge R(\chi_s s) \} \oplus \text{CSL}
\]

Here \(\mu_s s\) is a ghost component of type \(O = \{\text{ow}, \text{owm}\}\), 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 \(x \bullet \text{ow} = \text{ow} \bullet x = x\), so that \(\text{owm}\) is the unit of the operation. We leave \(\text{own} \bullet \text{own}\) 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 is pictorially shown in Figure 4, 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 \(\text{open}_\text{tr}\) and \(\text{close}_\text{tr}\) 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 \(\text{open}_\text{tr}\) with lock_tr, and \(\text{close}_\text{tr}\) with unlock_tr, so that the transitions execute simultaneously. This will give us an intermediate resource CSL’, and a morphism \(f : \text{Spin} \rightarrow \text{CSL'}\) deﬁned similarly as in Section II:

\[
\begin{align*}
\text{f}_\Sigma s &= s \downarrow 1 \\
\text{f}_\Delta s \text{ lock_tr} &= \text{lock_tr} \bowtie \text{open_tr} \\
\text{f}_\Delta s \text{ unlock_tr} &= \text{unlock_tr} \bowtie \text{close_tr}
\end{align*}
\]

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 \(\alpha_a\) and \(\chi_s\) used in these specs will be projections out of the state components of CSL.

However, if we try to carry out the above construction using the Spin resource from Section II, 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, \(f_\Delta s \text{ unlock_tr} = \text{unlock_tr} \bowtie \text{close_tr}\), 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 is 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 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 the left part of Figure 4. In particular, Spin should contain the integer components \(\pi_\alpha / \pi_\sigma\), which indicate if unlock_tr is “enabled for us” (\(\pi_\alpha s > 0\)), or not (\(\pi_\sigma s = 0\)), 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\(^{12}\). 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 following modiﬁcation of the specs from Section II, to indicate that lock enables

\(^{12}\)In general, the design of resource’s permissions obviously and essentially inﬂuences how that resource composes with others. Some systems, such as CAP [7] and iCAP [9], 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. For simplicity, we elide such generality, and consider only the permission to unlock, which sufﬁces to illustrate morphisms and simulations.
unlock_tr, and a successful unlock consumes one permission. Notice that having a permission to unlock doesn’t guarantee that it was “us” who last locked, or even that the lock is taken. We will impose such properties on CSL, but there is no need for them in Spin).

\[
\text{lock}^1 : [h, k]. \{(\lambda_s, \tau_s = h \land k \leq \text{last \_stamp}(\hat{\tau} s) \land \pi_s = 0) \}
\]
\[
\text{unlock}^1 : [h, k]. \{(\lambda_s, \tau_s = h \land k \leq \text{last \_stamp}(\hat{\tau} s) \land \pi_s = 1) \}
\]

Let us now consider the combination CSL’ of Spin and CSLX as defined in Figure 4. We see that the combination has a number of state components with overlapping roles. For example, \(\alpha\) from CSLX keeps the status of 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

\[
\omega(\tau_s s) = (\alpha_s s == \text{own})
\]
\[
\omega(\tau_o s) = (\alpha_o s == \text{own})
\]

(1)
as a basic coherency property.\(^{14}\) Furthermore, we want to encode exclusive locking, so we must require that only the thread that holds the lock has the permission to unlock:

\[
\pi_s s = (\text{if } \alpha_s s = \text{own then 1 else 0})
\]
\[
\pi_o s = (\text{if } \alpha_o s = \text{own then 1 else 0})
\]

(2)

(\text{thus, } \pi_s s, \pi_o s \in \{0, 1\}, \text{ and at most one of them is 1}). Also, the events recorded in the histories of Spin should correspond to exclusive locking, and thus:

\[
\tau_s s \perp_o \tau_o s
\]

(3)

where \(h \perp_o k\) is defined as

\[
(\text{last \_op} h = L \rightarrow \text{last \_stamp} k < \text{last \_stamp} (h)) \land
(\text{last \_op} k = L \rightarrow \text{last \_stamp} h < \text{last \_stamp} (k)) \land h \perp o k
\]
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.\(^{15}\)

It is now easy to see that \(J_1 = \{1\} \land \{2\} \land \{3\}\) is an invariant of CSL’. The critical point is that (3) is preserved by the \(t = \text{unlock\_tr} \triangleright \text{close\_tr}\) transition. Indeed, if in state \(s \in J_1\), we transition by \(t\), then we add a fresh unlocking entry to the ending state \(s’\), thus making last\_stamp \((\tau_s s’) > \) last\_stamp \((\tau_o s)\). For (3) to be preserved, it must then be last\_op \((\tau_o s’) = U\), i.e., the lock wasn’t held by another thread. But this is guaranteed by (2), as we can only have a permission to unlock in \(s \in J_1\), if we are the thread that last

13The specs for lock’ and unlock’ are small wrt. \(\pi_s s\), but large wrt. \(\tau_s s\).

14Strictly speaking, we should write \(\tau_s s (\langle 1 \rangle)\) (resp. \(\alpha_o s (\langle 2 \rangle)\)) to extract the self history (resp. other lock status) from the combined resource, as these come from the Spin (resp. CSLX) “sub-resource” of CSL’. However, as the components have distinct names, there is no danger of confusion which subresource they come from. We thus make the projections implicit and abbreviate the above values as \(\tau_s s\) and \(\alpha_o s\).

15Requirement (3) restricts only the last timestamp in \(h\) and \(k\), not all timestamps hereditarily, but that suffices for our proof.

locked. 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 \(J_1\) is invariant, we can construct a resource CSL out of CSL’, where \(J_1\) is imposed as an additional property of the underlying PCM and state space of CSL’. There is an elegant theory underlying this construction, which ensures that the set \(\Sigma(\text{CSL}) \land J_1\) can be made a global predicate, and thus be used as a state space of a new resource CSL\(^{[29]}\).

By Definition 2, globality depends on the underlying PCM, hence the construction involves restricting the PCM of CSL’ by \(J_1\). Technically this requires developing notions of sub-PCMs, PCM morphisms, and associated separating relations. For space reasons we elide the development, and refer the reader to \([29]\). Here, it suffices to say that the construction leads to the situation summarized by the following diagram:

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

where morphism \(\iota\) is defined by \(\iota_s s = s\) and \(\iota_s s = t\). Intuitively, CSL states are a subset of CSL’ states satisfying \(J_1\), and \(\iota_s s\) is the injection from \(\Sigma(\text{CSL})\) to \(\Sigma(\text{CSL’})\).

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:

\[
J_2 s \equiv \text{if } \alpha_s s = \text{own then } R(\sigma_s s) \text{ else } \sigma_s s = \emptyset
\]

Unlike \(J_1\), \(J_2\) is not an invariant, because it is perfectly possible for a thread to own the lock, but for its heap to not satisfy the predetermined property \(R\), because the thread has modified the acquired heap after locking it. However, \(J_2\) is an \((\iota \circ f)\)-simulation, as it satisfies the commuting diagrams from Figure 3. 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, \(J_2\) describes the state of CSL immediately after locking, and immediately before unlocking, which suffices for the morphing of lock’ and unlock’. We only show the derivation for exunlock = morph \((\iota \circ f)\) unlock’, and refer to the Coq code \([22]\) for the derivation of exlock.

1. \(\{\alpha_s s = \text{own} \land R(\sigma_s s)\}\)
2. \(\{\tau_s s = h \land \pi_s s = 1 \land \alpha_s s = \text{own} \land R(\sigma_s s)\}\)
3. \(\{\tau_s s = h \land k = \text{last\_stamp}(\hat{\tau} s) \land \pi_s s = 1 \land J_2 s\}\)
4. morph \((\iota \circ f)\) unlock’ // using simulation \(J_2\)
5. \(\{(\tau_s s = h \land t \Rightarrow U \land k < t \land \pi_s s = 0 \lor \tau_s s = h \land k \leq t \land \hat{\tau} s t = U \land \pi_s s = 1) \land J_2 s\}\)
6. \(\{\tau_s s = h \land t \Rightarrow U \land k < t \land \pi_s s = 0 \land J_2 s\}\)
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 must be false; that is, no other thread could have raced us to unlock before us in line 4. We infer that the disjunct is false by reasoning about the histories \(\tau_s s\) and \(\tau_o s\) in line 5. From \(\pi_s s = 1\), it must be \(\omega(\tau_s s) = \text{true}\), by the invariant \(J_1\) which holds throughout, as \(s\) is a CSL state. In particular, last\_op \((\tau_s s) = L\). By \(J_1\) again, \(\tau_s s\) and \(\tau_o s\) are
exclusive histories, so \( \text{last\_stamp}(\tau_o, s) < \text{last\_stamp}(\tau_s, s) \)
and it must be \( \text{last\_stamp}(\tau_s, s) = \text{last\_stamp}(\hat{\tau} s) = k \),
because \( \text{last\_stamp}(\hat{\tau} s) \) is the maximum of \( \text{last\_stamp}(\tau_s, s) \)
and \( \text{last\_stamp}(\tau_o, s) \). But this contradicts that \( \hat{\tau} s \) contains
entry \( U \) at \( t \geq k \). We now obtain the desired spec of ex unluck from
the beginning of the section, by taking \( \mu = \alpha \) and \( \chi = \sigma \).

V. QUIESCENCE AND INDEXED MORPHISM FAMILIES

The examples in Sections II and IV were examples of
abstract atomicity [11], as they were about “attaching” to the
Spin resource the functionality of another resource, Counter
and CSLX, with selected transitions of the two performing
simultaneously (i.e., atomically). In this section, we illustrate a
use of resource morphisms that goes beyond abstract atomicity, in
that it doesn’t involve combining resources, but showing how to “forget” the ghost state of one resource. 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 \( f_s : X \rightarrow \Sigma(W) \rightarrow \Sigma(V) \)
and the transition component \( f_t : X \rightarrow \Sigma(V) \rightarrow \Delta(V) \rightarrow \Delta(W) \)
now allow input \( X \), and specify a number of properties, listed
in Appendix B, which reduce to Definition 9 when \( X \) is
the unit type. Similarly, \( f \)-simulations must be indexed too,
and to be predicates over \( X \) and \( \Sigma(W) \), satisfying a number of
properties which reduce to Definition 10 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:

\[
\begin{align*}
\text{e} & : \{P \} A \{Q \} \oplus V \\
\text{morph } f x e & : \{\lambda s w. (f x)^\Sigma P s w \land I x s w \} A \\
& \{\lambda s w. \exists y. (f y)^\Sigma Q s w \land I y s w \} \oplus W
\end{align*}
\]

Here, \( (f x)^\Sigma P s w \) is defined as \( \exists s, s w = f_s x s w \land P s w \).

To illustrate, consider a spec for a concurrent stack’s push method, similar to that of lock from Section I:

\[
\begin{align*}
\text{push}(v) : [k]. & \{\lambda s. \sigma s = \emptyset \land \tau s = \emptyset \land k \leq \text{last\_stamp}(\tau o s) \} \\
& \{\lambda s. \sigma s = \emptyset \land \exists t v s, \tau s = t \Rightarrow (v s, v :: v s) \land \\
& k < t \} \oplus \text{Stack}
\end{align*}
\]

Here \( \tau_o \) and \( \tau_s \) are histories of stack’s operations, as in the
case of Spin, and \( \sigma o \) is the thread-private heap. The spec says
that push starts with \( \tau s = \emptyset \) (by framing, any history) and
ends with \( \tau_s = t \Rightarrow (v s, v :: v s) \) to indicate that a push
of \( v \) indeed occurred, and \( \forall t \) all the timestamps from the
pre-state. The Stack resource also has joint components.

The push method first allocates a new node in \( \sigma s \), then moves it to \( \sigma j \)
where it is enlinked to the top of the laid-out stack, after which
push updates \( \tau_j \) and \( \alpha_j \) to reflect the addition of the node. We
elide the full definition of Stack as it isn’t essential for the idea; it suffices to know that predicate layout describes how
\( \alpha_j \) is laid out in \( \sigma j \) (i.e., \( \forall s \in \Sigma(\text{Stack}), \text{layout}(\sigma_j s)(\sigma_j s) \)),
and that the global history \( \hat{\tau} s \) has no timestamp gaps (i.e.,
\( \forall s \in \Sigma(\text{Stack}), \text{dom}(\hat{\tau} s) = \{1, \ldots, \text{last\_stamp}(\hat{\tau} s)\} \)).

Consider now the program \( e = \text{push}(a) \parallel \text{push}(b) \) of the
following type, whose derivation is in the Coq files:

\[
\begin{align*}
\forall s. & \{\lambda s. \sigma s = \emptyset \land \tau s = \emptyset \} \\
& \{\lambda s. \sigma s = \emptyset \land \exists t j t_1 v s_1 t_2 v s_2. \\
& \tau s = t_1 \implies (v s_1, a :: v s_1) \parallel t_2 \implies (v s_2, b :: v s_2) \} \oplus \text{Stack}
\end{align*}
\]

The specification restricts that \( e \) pushes \( a \) and \( b \), to change the
stack contents from \( v s_1 \) to \( a :: v s_1 \) at time \( t_1 \), and from \( v s_2 \) to
\( b :: v s_2 \) at time \( t_2 \). The order of pushes is unspecified, so we
don’t know if \( t_1 < t_2 \) or \( t_2 < t_1 \) (as \( \parallel \) is commutative
and \( \tau \) and \( \tau_t \) in the binding to \( \tau s \) in the post doesn’t
imply an ordering between \( t_1 \) and \( t_2 \)). Moreover, we don’t
know that \( t_1 \) and \( t_2 \) occurred in immediate succession (i.e.,
\( t_2 = t_1 + 1 \lor t_1 = t_2 + 1 \)), as threads concurrent with \( e \)
could have executed between \( t_1 \) and \( t_2 \), changing the stack
arbitrarily. Thus, we also can’t infer that the ending state of
\( t_1 \) equals the beginning state of \( t_2 \), or vice versa.

But what if we knew that \( e \) is invoked without interfering
threads, i.e., quasiessently [10, 33–36]? For example, imagine
a resource \( \text{Priv} \) with only heaps \( \sigma s \) and \( \sigma o \), and no other
components, and transitions that allow modifying the self heap by
reading, writing, CAS-ing, or executing any other read-
modify-write command [32]10. A program working over \( \text{Priv} \)
can install an empty stack in \( \sigma s \) and then invoke \( e \) over it.
Because the stack is installed privately, no threads other
than the two children of \( e \) can race on it. Could we exploit
quiescence, and derive just out of the specification of \( e \) that
the stack at the end stores either the list \( [a, b] \), or \( [b, a] \)? This
fact can be stated even without histories, using solely heaps,
as follows:

\[
\{\lambda s. \text{layout nil}(\sigma s)\} \cup \{\lambda s. \text{layout}[a, b](\sigma s) \lor \\
\text{layout}[b, a](\sigma s)\} \oplus \text{Priv}
\]

The move from Stack to \( \text{Priv} \) 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
\( \text{Priv} \). We would like to obtain the \( \text{Priv} \) spec by applying
the morph rule to the Stack spec of \( e \), with a morphism
\( f : \text{Stack} \rightarrow \text{Priv} \) that forgets the histories. Unfortunately, such
a morphism can’t be constructed as-is. Were it to exist, then
\( f_\Sigma \), being contravariant, should map a state \( s_{\text{Stack}}, \) containing
only heaps, to a state \( s_{\text{Stack}}, \) containing heaps and histories; thus
\( f_\Sigma \) must “invent” the history component out of thin air.

This is where families come in. We make \( f : \text{Stack} \xrightarrow{\Sigma} \text{Priv} \)
a family over \( X = \text{hist} \), thereby passing to \( f_\Sigma \) the history \( x \)
that should be added to \( s_{\text{Stack}} \) to produce an \( s_{\text{Stack}} \).

\[
\begin{align*}
\forall x. s_{\text{Stack}} = s_{\text{Stack}} \equiv s_{\text{Stack}} \sigma s_{\text{Stack}} \sigma s_{\text{Stack}} \sigma j (s_{\text{Stack}}) \land \\
\tau s (s_{\text{Stack}}) = x \land \tau o (s_{\text{Stack}}) = \emptyset
\end{align*}
\]

16We implemented such a resource \( \text{Priv} \) in our Coq files [22]. It corresponds
to a separation logic with no ghost state.

17Such management of the scope of state is related to what the past work
has dubbed anti-framing [37, 38].
The first conjunct directly states that Stack is installed in \( \sigma \cdot (s'_{\text{priv}}) \) by making \( \sigma \cdot (s'_{\text{priv}}) \) be the join of the heaps \( \sigma_3 \cdot (s_{\text{stack}}) \) and \( \sigma_4 \cdot (s_{\text{stack}}) \). The second conjunct says that the heap \( \sigma_5 \cdot (s'_{\text{priv}}) \) of the interfering threads is propagated to \( \sigma \cdot (s_{\text{stack}}) \). 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 \( \tau \cdot (s_{\text{stack}}) \) history is declared \( \emptyset \), thus formalizing quiescence. We elide the definition of \( f \Delta \); 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-point operation such as read, write and CAS, but ignoring the ghost histories. Finally, applying the new morph rule with \( f \) to the Stack specification of \( e \), with \( I \cdot x \) being the always-true predicate on Priv states, and \( x = \emptyset \), gives us exactly the desired quiescent Priv spec. The full proof is in the Coq files, but briefly: let \( s'_{\text{priv}} \) be the state in the postcondition; then there exists \( s_{\text{stack}} \) and a history \( y \) such that \( \tau \cdot (s_{\text{stack}}) = y = t_1 \Rightarrow (u_{i1}, a :: u_{i2}) \circ \cdot t_2 \Rightarrow (u_{i2}, b :: u_{i3}) \), and \( \tau_0 \cdot (s_{\text{stack}}) = \emptyset \), and \( \sigma_4 \cdot (s'_{\text{priv}}) = \sigma_3 \cdot (s_{\text{stack}}) \). From \( \Sigma \cdot (\text{Stack}) \), we know layout \( (\alpha_3 \cdot (s_{\text{stack}})) \cdot (\sigma_3 \cdot (s_{\text{stack}})) \), and hence layout \( (\alpha_3 \cdot (s_{\text{stack}})) \cdot (\sigma_4 \cdot (s'_{\text{priv}})) \). Also from \( \Sigma \cdot (\text{Stack}) \), we know that \( \tau \cdot (s_{\text{stack}}) \) has no timestamp gaps; thus \( \{t_1, t_2\} = \text{dom} (\tau \cdot (s_{\text{stack}})) = \{1, 2\} \), i.e., \( t_1 \) and \( t_2 \) are the only, and consecutive, events. But then \( \sigma_3 \cdot (s_{\text{stack}}) \) must be either \( [a, b] \) or \([b, a]\), giving us the desired postcondition in Priv.

VI. RELATED WORK

Coalgebra morphisms. State transition systems are mathematically representable as coalgebras [39, 40]; thus, resource morphisms are closely related to coalgebra morphisms. There are, however, differences between our definition of morphisms and the coalgebras, arising from our application to concurrent separation logic. For example, for us, given \( f : V \rightarrow W \), \( f \Sigma \) is contravariant, but in the coalgebraic case, \( f \) is covariant on states [39, 41]. A coalgebra morphism \( f \) doesn’t have the \( f \Delta \) component, but requires that \( f \) preserves and reflects 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 10 of \( f \)-simulations. Moreover, we show how \( f \) acts on programs, and provide a Hoare logic rule relating the morphed programs and simulations.

Abstract atomicity. Jacobs and Piessens [18] have originated an approach to abstract atomicity, later expanded by Svendsen et al. in HOCAP [8] and iCAP [9], and by Jung et al. in IRIS [12, 19], that relies on parametrizing a program and its proof with ghost code that works over the state of other resources. For example, using names from Sections II, the

18 As we want to build \( s_{\text{stack}} \) out of \( s'_{\text{priv}} \), we have to identify the part of \( \sigma_4 \cdot (s'_{\text{priv}}) \) which we want to assign to \( \sigma_3 \cdot (s_{\text{stack}}) \). This part has to be uniquely determined, else \( f \Sigma \) won’t be a function. We ensure uniqueness by insisting that the predicate layout is precise – a property commonly required in separation logics.

19 In our Coq files, we carried out this development for a Treiber variant of concurrent stacks, with some minor Treiber-specific modifications.

program lock over Spin can be parametrized by a ghost function over Counter that increments the counter at the moment of successful locking, much like our incr_tr transition in Figure 2. In contrast, we formalized the scenario in Section II by exhibiting a morphism from Spin to the extended resource SC that combines Spin with Counter.

The parametrization affords abstract higher-order specifications that are similar to specifications that one would ascribe to a data structure in the sequential setting (and, in particular, don’t require specifications by histories). One known challenge in the case of parametrization, however, arises when the point at which to execute the ghost function can be determined only after the program has already terminated. This is a common pattern when proving linearizability of concurrent programs, as exhibited, say, by the queue of Herlihy and Wing [1], but can’t directly be addressed by parametrization, because it is unclear at which point in the code to invoke the ghost function.

Using histories for specification enables separating the tracking of termination and ownership of operations from their order in the linearization, as this order becomes just another ghost component that can be constructed at run-time, with cooperation of other threads [42]. A spec can then say that a method finished executing some operation, but that the exact place of the operation in the linearization is to be fully determined only later, by the action of other threads. We expect that our history specifications and morphism will scale to such examples, but this remains future work.

Da Rocha Pinto et al. [11] in TADA logic, introduce a new judgment form, \( (P, e @ Q) \), capturing 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 \( e \) makes 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 contrast, we specify programs using ordinary Hoare triples, but rely on the PCM of histories to express atomicity (i.e., 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 [24], where one thread executes work on behalf of others; helping is an idiom that TADA can’t currently express.

IRIS [12] features a rule \( \text{INV} \), which is similar to our morph rule. Crucially, however, \( \text{INV} \), and several related rules, are restricted to atomic programs only—a restriction used in [12] to motivate the parametrization approach to abstract atomicity. In contrast, our morph rule applies uniformly to all programs, irrespective of atomicity. Doing so required developing morphisms, and clarifying the connection between abstract atomicity and simulations. For example, one of the key notions in IRIS is invariants, whereas for us, invariants are a special case of \( f \)-simulations, when \( f = \text{id} \). Our development also made the theory mathematically and conceptually simple. For example, we were able to elide numerous technical concepts that arise in IRIS, such as invariant name spaces, invariant masks, cores of PCM elements, and many modalities. On the
other hand, the model of IRIS supports impredicative higher-order state, which our model (Appendix C) doesn’t. In the future, we would like to develop a higher-order model, but we expect that such a model won’t alter the algebraic concepts, the logic, and the reasoning principles presented here.

We noted that resource morphisms go beyond abstract atomicity, as they apply to quiescence (Section V), which is typically not considered related to abstract atomicity. Also, resource morphisms can apply to categorical questions such as e.g., “when are two resources equivalent” (answer: when there are cancelling morphisms between them), which we plan to explore in the future.

Protocol hooks. Sergey et al. [43] have designed a logic DISEL for distributed systems, in which one can combine distributed protocols—represented as state transition systems—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. id_tr P = λs s'. P s∧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.

Linearizability and refinement. In a somewhat different, relational, flavor of separation logics [16, 17, 44], and more generally, in the work on proving refinement and linearizability [45–51], 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 this approach over refinement and linearizability. First, a spec in the form of a Hoare triple enriched 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 refinement, one also has to verify the refined program itself. Second, linearizability has traditionally had difficulties in addressing ownership transfer of heaps between data structures [52, 53], whereas for us (and other extensions of CSL [7, 11, 12, 19, 46]), 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. Finally, our specs of lock and unlock in Section II 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. However, as shown by Sergey et al. [36], the PCM-based approach can be more general and is capable of proving correctness of useful non-linearizable programs as well. In future work we plan to apply resource morphisms to such proofs.

FCSL. The current paper adds to FCSL [10] the novel notions of resource morphism, and significantly modifies the notion of resources. In old FCSL, each concurrent resource was a finite map from labels (natural numbers) to sub-components. For example, using the concepts from Section II, one could represent SC as a finite map l₁ ⇒ Spin ⊔ l₂ ⇒ Counter, where l₁ and l₂ 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 ⊔. 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 tuples which are combined by pairing (Appendix A). Consequently, if we changed the definition of SC in Section II 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 the technical contributions of this paper. FCSL previously supported quiescence by means of a dedicated and very complex inference rule, whereas Section V demonstrates that quiescence is merely an application of morphism families.

VII. 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.

A morphism from resource V to resource W exhibits a form of forward simulation [54] of V by W, and is stable, i.e., it also exhibits a form of (weak) simulation of W by V on transposed states. Morphisms and simulations are integrated into separation logic via a single inference rule that propagates the simulation from the precondition to the postcondition.

Resource 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 II, is specialized to a specific ownership discipline, e.g., exclusive locking in Section IV (though we have also developed [29] an instantiation to readers-writers [26] locking). Other applications include the managing of scope of ghost state in quiescent environments, as illustrated in Section V, and restricting the state space of a resource with additional state and PCM invariants [29].
REFERENCES


H. Liang, X. Feng, and M. Fu, “A rely-guarantee-based


APPENDIX A  

PCM PRODUCTS, TRANSITION COUPLINGS AND RESOURCE COMBINATIONS

In this section we develop the combination of two resources (the combined resource or tensor resource) that has been used in Sections II and IV. For that, we need first to give some basic definitions for the product of PCMs and the coupling of transitions.

**Lemma 12.** Let $A$ and $B$ be PCMs. Then, the Cartesian product $A \times B$ is a PCM under the point-wise lifting of the join operators, $(a_1, b_1) \bullet_{A \times B} (a_2, b_2) = (a_1 \bullet_A a_2, b_1 \bullet_B b_2)$, and units, $1_{A \times B} = (1_A, 1_B)$.

For example, the PCM for SC in Figure 2 is the product of the PCMs $M(\text{Spin})$ and $M(\text{Counter})$, that is, $M(\text{SC}) = M(\text{Spin}) \times M(\text{CSLX})$.

**Definition 13.** Let $s_i = (a^i_1, a^i_2, a^i_3)$ be an $(M_i, T_i)$-state, $i = 1, 2$, and $s = (a_s, a_j, a_o)$ be an $(M_1 \times M_2, T_1 \times T_2)$-state. The pair state of $s_1$ and $s_2$ is $[s_1, s_2] \in \{(a^1_1, a^2_1), (a^1_2, a^2_2), (a^1_3, a^2_3)\}$, and projection states of $s$ are $s^i = (a^i_1, a^i_2, a^i_3)$, $i = 1, 2$. The usual beta and eta laws for products hold, i.e.: $[s_1, s_2]|i = s_i$ and $s = [s_1, s_2]|2$.

**Definition 14.** Given state spaces $\Sigma_1$ and $\Sigma_2$, the product state space $\Sigma_1 \times \Sigma_2$ is defined by:

$$\begin{align*}
\text{s} \in \Sigma_1 \times \Sigma_2 & \iff s^1 \in \Sigma_1 \land s^2 \in \Sigma_2 \land \exists \Sigma^s_1 \land \Sigma^s_2 \\
\text{s} \in \Sigma_1 \times \Sigma_2 & \iff s \in \Sigma \land s \in \Sigma \land \exists \Sigma^s_1 \land \Sigma^s_2 \\
\text{I} & = \{(\text{lock_tr}, \text{open_tr}), (\text{unlock_tr}, \text{close_tr}), (\text{id_tr}, \text{id_tr})\}
\end{align*}$$

Here, id is the idle transition defined as $\lambda s \cdot s' = s$. We have implicitly assumed that all resources in this paper possess an idle transition, which is why we elided this transition from the figures.

**APPENDIX B**

INDEXED MORPHISM FAMILIES

In this appendix, we show how the definitions of morphism and $f$-simulations generalize to indexed families. When $X$ is the unit type, we recover the morphism-related definitions from Section III.

**Definition 17.** An indexed family of morphisms $\text{id} : V \to \Sigma(W)$ (or just family), consists of partial functions $f_{\Sigma} : X \to \Sigma(W) \to \Sigma(V)$ (note the contravariance), and $f_{\Delta} : X \to \Delta(V) \to \Delta(W)$ on transitions, such that:

1. (locality of $f_{\Sigma}$) there exists a function $f_{\Delta} : M(V) \to M(W)$ such that if $f_{\Sigma} (s_w \triangleright p) = s_v$, then $s_w = s'_v \triangleright f_{\Delta} (p)$, and $f_{\Sigma} (s_w \triangleleft p) = s'_v \triangleleft f_{\Delta} (p)$.

2. (locality of $f_{\Delta}$) if $f_{\Delta} (x \geq s_w \triangleright p) = t'$, then $f_{\Delta} (x \geq s_w \leq p) = t = t'$. 

3. (other-fixity) if $a_o (s_w) = a'_o (s'_w)$ and $f_{\Sigma} x s_w$, $f_{\Sigma} x' s'_w$ exist, then $a_o (f_{\Sigma} x s_w) = a_o (f_{\Sigma} x' s'_w)$.

4. (injectivity on indices) if $f_{\Sigma} x s_{w_1}$ and $f_{\Sigma} x s_{w_2}$ exist and $f_{\Sigma} x s_{w_1} = f_{\Sigma} x s_{w_2}$, then $x = x_2$. In most of the properties of Definition 17, the index $x$ is simply propagated unchanged. The only property where $x$ is significant is the new property (4), which requires that $f_{\Sigma} x s_{w}$ uniquely determines the index $x$. In the Stack example in Section V, it is easy to see that the definition of $f_{\Sigma}$ satisfies this property, because equal states have equal histories.

**Definition 18.** Given a morphism family $f : V \to W$, an $f$-simulation is a predicate $I$ on $X$ and $W$-states such that:

1. if $I x s_w$, and $f_{\Sigma} x s_w = s_v$, and $t v s'_v$, then there exist $x', t' = f_{\Delta} x s_w t$ and $s'_w$ such that $f_{\Sigma} x' s'_w$.

2. if $I x s_w$, and $s_w = f_{\Sigma} x s_w$ exists, and $s_w \triangleright \text{eval}^x s'_w$, and

3. if $I x s'_w$, and $s'_w = f_{\Sigma} x s'_w$ exists, and $s'_w \triangleright \text{eval}^x s'_w$.

Compared to Definition 10, property (1) allows that $x$ changes into $x'$ by a transition. In the Stack example in Section V, if we lift $e$ by using the index $x = \text{empty}$ (i.e., write morph empty $f e$), then this index will evolve with $e$ taking the transitions of Stack to track how $e$ changes the self history by adding the entries for pushing 1 and 2. In property (2), the index $x$ simply propagates.

**APPENDIX C**

DENOTATIONAL SEMANTICS

Our semantic model largely relies on the denotational semantic of action trees [27]. A tree implements a finite partial approximation of one execution of a program. Thus, a program of type $\text{ST} V A$ will be denoted by a set of such trees. The set may be infinite, as some behaviors (i.e., infinite loops) can only be represented as an infinite set of converging
approximated executions. Because a program may contain multiple executions, our denotational semantics can represent non-determinism in the form of internal choice, though our language currently does not make use of that feature.

An action tree is a generalization of the Brookes' notion of action trace [6] in the following sense. Where action trace semantics represent a program by a set of traces, we represent a program with a set of trees. A tree differs from a trace in that a trace is a sequence of actions and their results, whereas a tree contains an action followed by a continuation which itself is a tree parameterized wrt. the output of the action.

In this appendix, we first define the denotation of each of our commands as a set of trees. Then we define the semantic behavior for trees wrt. resource states, in a form of operational semantics for trees, thus formalizing single execution of a program. Then we relate this low-level operational semantics of trees to high-level transitions of a resource by an always predicate (Section C-C) that ensures that a tree is resilient to any amount of interference, and that all the operational steps by a tree are safe. The always predicate will be instrumental in defining the vrf-predicate transformer from Section III, and from there, in defining the type of Hoare triples \{P\} A \{Q\} @V. Both the ST V A type and the Hoare triple type will be complete lattices of sets of trees, giving us a suitable setting for modeling recursion. The soundness of FCSL follows from showing that the lemmas about the vrf predicate transformer listed in Section III, are satisfied by the denotations of the commands.

We choose the Calculus of Inductive Constructions (CiC) [55, 56] as our meta logic. This has several important benefits. First, we can define a shallow embedding of our system into CiC that allows us to program and prove directly with the semantic objects, thus immediately lifting to a full-blown programming language and verification system with higher-order functions, abstract types, abstract predicates, and a module system. We also gain a powerful dependently-typed \lambda-calculus, which we use to formalize all semantic definitions and meta theory, including the definition of action trees by iterated inductive definitions [55], specification-level functions, and programming-level higher-order procedures. Finally, we were able to mechanize the entire semantics and meta theory in the Coq proof assistant implementation of CiC.

A. Action trees and program denotations

**Definition 19** (Action trees). The type tree V A of (A-returning) action trees is defined by the following iterated inductive definition.

\[
\text{tree } V A \triangleq \text{Unfinished} \\
\quad | \text{Ret} (v : A) \\
\quad | \text{Act} (a : \text{action } V A) \\
\quad | \text{Seq} (T : \text{tree } V B) (K : B \to \text{tree } V A) \\
\quad | \text{Par} (T_1 : \text{tree } V B_1) (T_2 : \text{tree } V B_2) \\
\qquad (K : B_1 \times B_2 \to \text{tree } V A) \\
\quad | \text{Morph} (f : W X \to V) (x : X) (T : \text{tree } W A)
\]

Most of the constructors in Definition 19 are self-explanatory. Since trees have finite depth, they can only approximate potentially infinite computations, thus the Unfinished tree indicates an incomplete approximation. Ret v is a terminal computation that returns value v : A. The constructor Act takes as a parameter an action a : action V A, as defined in Section III. Seq T K sequentially composes a B-returning tree T with a continuation K that takes T's return value and generates the rest of the approximation. Par T_1 T_2 K is the parallel composition of trees T_1 and T_2, and a continuation K that takes the pair of their results when they join. CiC's iterated inductive definition permits the recursive occurrences of tree to be nonuniform (e.g., tree B_i in Par) and nested (e.g., the positive occurrence of tree A in the continuation). Since the CiC function space includes case-analysis, the continuation may branch upon the argument. The Morph constructor embeds an index x : X, morphism f : W X \to V, and tree T : tree W A for the underlying computation. The constructor will denote T should be executed so that each of its actions is modified by f with an index x. We can now define the denotational model of our programs; that is the type ST V A of sets of trees, containing Unfinished.

\[
\text{ST } V A \triangleq \{ e : \text{set } (\text{tree } V A) | \text{Unfinished } e \}
\]

The denotations of the various constructors combine the trees of the individual denotations, as shown below.

\[
\begin{align*}
\text{ret } (r : A) & \triangleq \{ \text{Unfinished}, \text{Ret } r \} \\
x & \leftarrow e_1; e_2 & \triangleq \{ \text{Unfinished} \} \cup \{ \text{Seq } T_1 K | T_1 \in e_1 \land \forall x. K x \in e_2 \} \\
e_1 \parallel e_2 & \triangleq \{ \text{Unfinished} \} \cup \{ \text{Par } T_1 T_2 \text{ Ret } | T_1 \in e_1 \land T_2 \in e_2 \} \\
(a) & \triangleq \{ \text{Unfinished}, \text{Act } a \} \\
\text{morph } f x e & \triangleq \{ \text{Unfinished} \} \cup \{ \text{Morph } f x T | T \in e \}
\end{align*}
\]

The denotations of ret simply contains the trivial Ret tree, in addition to Unfinished, and similarly in the case of (a). The trees for sequential composition of e_1 and e_2 are obtained by pairing up the trees from e_1 with those from e_2 using the Seq constructor, and similarly for parallel composition and morphism application.

The denotations of composed programs motivate why we denote programs by non-empty sets, i.e., why each denotation contains at least Unfinished. If we had a program Empty whose denotation is the empty set, then the denotation of x \leftarrow Empty; e', Empty \parallel e' and morph f x Empty will all also be empty, thus ignoring that the composed programs exhibit more behaviors. For example, the parallel composition Empty \parallel e' should be able to evaluate the right component e', despite the left component having no behaviors. By including Unfinished in all the denotations, we ensure that behaviors of the components are preserved in the composition. For example, the parallel composition \{Unfinished\} \parallel e' is denoted by the set:

\[
\{ \text{Unfinished} \} \cup \{ \text{Par Unfinished } T \text{ Ret } | T \in e' \}
\]

which contains an image of each tree from e', thus capturing the behaviors of e'.

16
\[ \Delta \vdash x, s, \text{Seq} \ (\text{Ret} \ v) \ K \xrightarrow{\text{SeqRet}} \bar{x}, s, K \ v \]
\[ \Delta \vdash x, s, T \xrightarrow{\pi} \bar{x}, s', T' \]
\[ \Delta \vdash x, s, \text{Seq} \ T \ K \xrightarrow{\text{SeqStep} \ (\pi : \text{path})} \bar{x}', s', \text{Seq} \ T' \ K \]

\[ \Delta \vdash x, s, \text{Par} \ (\text{Ret} \ v_1) \ (\text{Ret} \ v_2) \ K \xrightarrow{\text{ParRet}} \bar{x}, s, K \ (v_1, v_2) \]
\[ \Delta \vdash x, s, \text{Par} \ T_1 \ T_2 \ K \xrightarrow{\text{ParL} \ (\pi : \text{path})} \bar{x}', s', \text{Par} \ T_1' \ T_2' \ K \]
\[ \Delta \vdash x, s, \text{Par} \ T_1 \ T_2 \ K \xrightarrow{\text{ParR} \ (\pi : \text{path})} \bar{x}', s', \text{Par} \ T_1 \ T_2' \ K \]
\[ \text{unwind} \ _\text{act} \ \Delta \bar{x}' \ a \ s' \ v \]
\[ \Delta \vdash x, s, \text{Act} \ a \xrightarrow{\text{ActChoice}} \bar{x}', s'; \text{Ret} \ v \]
\[ \Delta \vdash x, s, \text{Morph} \ f \ y \ (\text{Ret} \ v) \xrightarrow{\text{MorphRet}} \bar{x}, s, \text{Ret} \ v \]
\[ \Delta \vdash x, s, \text{Morph} \ f \ y \ T \xrightarrow{\text{MorphStep} \ (\pi : \text{path})} \bar{x}', s', \text{Morph} \ f \ y' \ T' \]

Fig. 5: Judgment \( \Delta \vdash x, s, T \xrightarrow{\pi} \bar{x}', s', T' \), for operational semantics on trees, which reduces a tree with respect to the path \( \pi \).

B. Operational semantics of action trees

The judgment for small-step operational semantics of action trees has the form \( \Delta \vdash x, s, T_v \xrightarrow{\pi} \bar{x}', s', T_v' \) (Figure 5). We explain the components of this judgment next.

First, the component \( \Delta \) is a morphism context. This is a sequence, potentially empty, of morphism families

\[ f_0 : V_1 \xrightarrow{\delta} W, f_1 : V_2 \xrightarrow{\delta} V_1, \ldots, f_n : V \xrightarrow{\delta} V_n \]

We say that \( \Delta \) has resource type \( V \rightarrow W \), and index type \( (X_0, \ldots, X_n) \). An empty context \( \cdot \) has resource type \( V \rightarrow V \) for any \( V \).

Second, the components \( \bar{x} \) and \( \bar{x}' \) are tuples, of type \( (X_0, \ldots, X_n) \), and we refer to them as indexes. Intuitively, the morphism context records the morphisms under which a program operates. For example, if we wrote a program of the form

\[ \text{morph} \ f_0 \ x_0 \ (\cdots (\text{morph} \ f_n \ x_n \ c) \cdots) \]

it will be that the trees that comprise \( e \) execute under the morphism context \( f_0, \ldots, f_n \), with an index tuple \( (x_0, \ldots, x_n) \).

Third, the components \( s_w \) and \( s_w' \) are \( W \)-states, and \( T_v, T_v' \) : tree \( V \ A \), for some type \( A \). The meaning of the judgment is that a tree \( T_v \), when executed in a state \( s_w \), under the context of morphisms \( \Delta \) produces a new state \( s_w' \) and residual tree \( T_v' \), encoding what is left to execute. The resource of the trees and the states disagree (the states use resource \( W \), the trees use \( V \)), but the morphism context \( \Delta \) relates them as follows. Whenever the head constructor of the tree is an action, the action will first be morphed by applying all the morphisms in \( \Delta \) in order, to the transitions that constitute the head action, supplying along the way the projections out of \( x \) to the morphisms. This will produce a new index \( x' \) and an action on \( W \)-states, which can be applied to \( s_w \) to obtain \( s_w' \).

Fourth, the component \( \pi \) is of path type, identifying the position in the tree where we want to make a reduction.

The key are the constructors \( \text{ParL} \ \pi \) and \( \text{ParR} \ \pi \). In a tree which is a \( \text{Par} \) tree, these constructors identify that we want to reduce in the left and right subtree, respectively, iteratively following the path \( \pi \). If the tree is not a \( \text{Par} \) tree, then \( \text{ParL} \) and \( \text{ParR} \) constructors will not form a good path; we define further below when a path is good for a tree. The other path constructors identify positions in other kinds of trees. For example, \( \text{ChoiceAct} \) identifies the head position in the tree of the form \( \text{Act}(a) \), \( \text{SeqRet} \) identifies the head position in the tree of the form \( \text{Seq} \ (\text{Ret} \ v) \ K \) (i.e., it identifies a position of a beta-reduction), \( \text{SeqStep} \ \pi \) identifies a position in the tree \( \text{Seq} \ T \ K \), if \( \pi \) identifies a position within \( T \), etc. We do not paths for trees of the form \( \text{Unfinished} \) and \( \text{Ret} \ v \), because these do not reduce.

In order to define the operational semantics on trees, we next require a few auxiliary notions. First, we need the relation \( \text{ComSquares} \ \Delta \bar{x} \bar{x}' s_w s'_w t_v s_w' s_w'' t_w \) encoding the composition of the commutative diagrams below (each commutative diagram is an instance of Figure 2(b), for one of the morphisms \( f_0, f_1, \ldots, f_n \), in the context \( \Delta = f_0; f_1; \cdots; f_n \)).

The relation captures that \( t_w \) is morphed by iterating the state component of the morphisms in \( \Delta \), starting from the state \( s_w \), and passing along the elements of the tuple \( \bar{x} \) or type \( (X_0, \ldots, X_n) \), until we reach the state \( s_w' \). Then transition \( t_w \) is executed in \( s_w \) to obtain \( s_w' \), and the transition components of the morphisms are run backwards to obtain \( t_w' \), and the associated end state \( s_w'' \). Formally:

\[ \text{ComSquares} \ (\cdot) (\cdot) \ s_w' s_w t_v s_w' s_w'' t_w \equiv \\
\]
\[ s_w = s_w' \land s_w = s_w'' \land t_w = t_w' \]

\[ \exists s_1 t_1. \text{ComSquares} \ \Delta \bar{x} \bar{x}' s_w' s_w t_v s_1 s_1' t_1 \land \\
\]
\[ f_0\delta x_0 x_w = s_1 \land f_0\delta x_0 x_w = s_1 \]

17
We will often abbreviate ComSquares, and write:

\[
\text{unwind}_\text{tr} \Delta \Delta \preceq x \bar{x} \bar{t} v s_w s_w' \equiv \\
\exists s_v \bar{s}_v \bar{t}_v \text{ComSquares} \Delta \Delta \preceq x \bar{x} \bar{t} v s_w s_w' \equiv \\
\exists s_v \bar{s}_v \bar{t}_v \Delta \Delta \preceq x \bar{x} \bar{t} v s_w s_w' \equiv \\
\exists s_v \bar{s}_v \bar{t}_v, a \text{ s.t. } s_v = (t_v, v) \wedge \\
\text{ComSquares} \Delta \Delta \preceq x \bar{x} \bar{t} v s_w s_w' \equiv \\
\]

Given a context \( \Delta \), input indices \( \bar{x} \), input state \( s_w \) and input transition \( t_v \), \text{unwind}_\text{tr} “undoes” the transition, in the sense that its computes the output index \( \bar{x} \), output state \( s_w' \), obtained by the morphing iteratively of the morphisms in \( \Delta \). The abbreviation \text{unwind}_\text{act} does the same, but for input action \( a \) : actionV.A. In the process, \text{unwind}_\text{act} also computes the return value \( v \).

In the frequent cases where \( \Delta \) is the empty context and (correspondingly, \( \bar{x} \) and \( \bar{x}' \) are empty tuples ()), we will abbreviate the judgment from Figure 5, and simply write \( s, T \Delta \equiv v, s', T' \).

The operational semantics on trees in Figure 5 may not make a step on a tree for two different reasons. The first, benign, reason is that the chosen path \( \pi \) does not actually determine an action or a redex in the tree \( T \). For example, we may have \( T = \text{Unfinished} \) and \( \pi = \text{ParR} \). But we can choose the right side of a parallel composition only in a tree whose head constructor is Par, which is not the case with Unfinished. We consider such paths that do not determine an action or a redex in a tree to be ill-formed. The second reason arises when \( \pi \) is actually well-formed. In that case, the constructors of the path uniquely determine a number of rules of the operational semantics that should be applied to step the tree. However, the premises of the rules may not be satisfied. For example, in the ChoiceAct rule, there may not exist a \( v \) such that \text{unwind}_\text{tr} \Delta \Delta \preceq x \bar{x}' a s s' v. To differentiate between these two different reasons, we first define the notion of well-formed, or good path, for a given tree.

\textbf{Definition 20 (Good paths).} Let \( T : \text{tree} \; V \; A \) and \( \pi \) be a path. Then the predicate good \( T \pi \) recognizes good paths for a tree \( T \) as follows:

- good (Act \( a \) ) ChoiceAct \( \equiv \) true
- good (Seq (Ret \( v \) ) _) SeqRet \( \equiv \) true
- good (Seq \( T _ - _ _ ) \) SeqRet \( \equiv \) true
- good (Par (Ret _ _ _ ) ) ParRet \( \equiv \) true
- good (Par \( T _ - _ _ ) \) ParRet \( \equiv \) true
- good (Morph f x (Ret _ _ _ ) ) MorphRet \( \equiv \) true
- good (Morph f x T ) MorphStepPar \( \equiv \) true
- good T \( \equiv \) false otherwise

\textbf{Definition 21 (Safe configurations).} We say that a state \( s \) is safe for the tree \( T \) and path \( \pi \), written \( s \in \text{safe} \; t \pi \) if:

- good \( T \pi \rightarrow \exists s' T', s, T \Delta \rightarrow s', T' \)

Notice that in the above definition, the trees Unfinished and Ret \( v \) are safe for any path, simply because there are no good paths for them, as such trees are terminal. On the other hand, a tree Act \( a \) does have a good path, namely ChoiceAct, but may be unsafe, if the action \( a \) is not defined on input state \( s \).

For example, the \( a \) may be an action for reading from some pointer \( x \), but that pointer may not be allocated in the state \( s \).

Safety of a tree will be an important property in the definition of Hoare triples, where we will require that a precondition of a program implies that the trees comprising the program’s denotation are safe for every path.

The following are several important lemmas about trees and their operational semantics, which lift most of the properties of transitions, to trees.

\textbf{Lemma 22 (Coverage of stepping by transitions).} Let \( \Delta : V \rightarrow W \), and \( \Delta \Delta \preceq x, s, T \rightarrow s', T', \) Then either the step corresponds to an idle transition, that is, \( (\bar{x}, s) = (x, s') \), or there exists a transition \( a \in \Delta(V) \), such that \text{unwind}_\text{tr} \Delta \Delta \preceq x \bar{x}' a s s' v.

\textbf{Lemma 23 (Other-fixity of stepping).} Let \( \Delta : V \rightarrow W \) and \( \Delta \Delta \preceq x, s, T \rightarrow s', T' \). Then \( a_0(s) = a_0(s') \).

\textbf{Lemma 24 (Stepping preserves state spaces).} Let \( \Delta : V \rightarrow W \) and \( \Delta \Delta \preceq x, s, T \rightarrow s', T' \). If \( s \in \Sigma(W) \) then \( s' \in \Sigma(W) \).

\textbf{Lemma 25 (Stability of stepping).} Let \( \Delta : V \rightarrow W \) and \( \Delta \Delta \preceq x, s, T \rightarrow s', T' \). Then \( s' \rightarrow \rightarrow s' s' T \).

\textbf{Lemma 26 (Determinism of stepping).} Let \( \Delta : V \rightarrow W \) and \( \Delta \Delta \preceq x, s, T \rightarrow s', T' \). Then \( a(s) = a(s') \).

\textbf{Lemma 27 (Locality of stepping).} Let \( \Delta : V \rightarrow W \) and \( \Delta \Delta \preceq x, s, T \rightarrow s', T' \). Then \( \exists s'' \text{ such that } s'' \rightarrow p \in \text{safe} \; T \).

\textbf{Lemma 28 (Safety-monotonicity of stepping).} If \( s \rightarrow p \in \text{safe} \; T \pi \text{ then } s \rightarrow p \in \text{safe} \; T \pi \).

\textbf{Lemma 29 (Frameability of stepping).} Let \( s \rightarrow p \in \text{safe} \; T \pi \), and \( s \rightarrow p \in \text{safe} \; T \pi \). Then \( \exists s'' \text{ such that } s'' \rightarrow p \in \text{safe} \; T \pi \).

The following lemma is of crucial importance, as it relates stepping with morphisms. In particular, it says that the steps of a tree are uniquely determined, no matter the morphism under which it appears. Intuitively, this holds because each transition that a tree makes has a unique image under a morphism \( f : V \rightarrow W \).

\textbf{Lemma 30 (Stepping under morphism).} Let \( f : V \rightarrow WX \) and \( f x s_w = s_v \). Then the following hold:

1) \( f \rightarrow s_w \rightarrow s'_w \rightarrow T', \) then \( \exists x' s'_w \rightarrow s'_v \rightarrow f \rightarrow \rightarrow (x', s'_w, T') \).

2) \( f \rightarrow \rightarrow (x, s_w) \rightarrow (x', s'_w, T') \). Then \( \exists x' s'_w \rightarrow s'_v \rightarrow f \rightarrow \rightarrow (x', s'_w, T') \).

The first property of this lemma relies on the fact that for a step over states in \( V \), we can also find a step over related states in \( W \), i.e., that \( f \) encodes a simulation. The second property relies on the fact that \( f \)’s state component is a function in the contravariant direction. Thus, for each \( s_w \) there are unique \( x \) and \( s_v \), such that \( f x s_w = s_v \).
C. Predicate transformers

In this section we define a number of predicate transformers over trees that ultimately lead to defining the vrf predicate transformer on programs.

Definition 31 (Modal Predicate Transformers). Let $T : \text{tree } V A$, and $\zeta$ be a sequence of paths. Also, let $X$ be an assertion over $V$-states and $V$-trees, and $Q$ be an assertion over $A$-values and $V$-states. We define the following predicate transformers:

$$\text{always}^\zeta T X s \triangleq \begin{cases} & \text{if } \zeta = \pi \vdash \zeta' \text{ then } \\ & \forall s_2. (s \xrightarrow{\pi} s_2) \rightarrow \text{safe } T \pi s_2 \land X s_2 T \land \\ & \forall s_3. T''. (s_2, T' \xrightarrow{s_3}, T') \rightarrow \text{always}^{\zeta''} T_2 X s_3 \\ & \text{else } \forall s_2. (s \xrightarrow{\pi} s_2) \rightarrow X s_2 T \\ \text{always } T X s \triangleq \forall \zeta, \text{always}^\zeta T X s \\ \text{after } T Q \triangleq \text{always } T (\lambda s'. T'. \forall v. T' = \text{Ret } v \rightarrow Q v s')$$

The helper predicate always$^\zeta T X s$ expresses the fact that starting from the state $s$, the tree $T$ remains safe and the user-chosen predicate $X$ holds of all intermediate states and trees obtained by evaluating $T$ in the state $s$ according to the sequence of paths $\zeta$. The predicate $X$ remains valid under any any environment steps of the resource $V$.

The predicate always $T X s$ quantifiers over the path sequences. Thus, it expresses that $T$ is safe and $X$ holds after any finite number of steps which can be taken by $T$ in $s$.

The predicate transformer after $T Q$ encodes that $T$ is safe for any number of steps; however, $Q v s'$ only holds if $T$ has been completely reduced to Ret $v$ and state $s'$. In other words $Q$ is a postcondition for $T$, as it is required to hold only if, and after, $T$ has terminated.

Now we can define the vrf predicate transformer on programs, by quantifying over all trees in the denotation of a program.

$$\text{vrf } e Q s \triangleq \forall T \in e. \text{after } T Q s$$

This immediately gives us a way to define when a program $e$ has a precondition $P$ and postcondition $Q$: when all the trees in $T$ have a precondition $P$ and postcondition $Q$ according to the after predicate, or equivalently, when

$$\forall s \in \Sigma (V). P s \rightarrow \text{vrf } e Q s$$

which is the formulation we used in Section III to define the Hoare triples.

We can now state the following soundness theorem, each of whose three components has been established in the Coq files.

Theorem 32 (Soundness).

- All the properties of vrf predicate transformer from Section III are valid.
- The sets $\mathcal{ST} V A$ and $\{P\} A \{Q\}$ are complete lattices under subset ordering with the set $\{\text{Unfinished}\}$ as the bottom. Thus one can compute the least fixed point of every monotone function by Knaster-Tarski theorem.