Specifying Concurrent Programs in Separation Logic: Morphisms and Simulations

ALEKSANDAR NANEVSKI, IMDEA Software Institute, Spain
ANINDYA BANERJEE, IMDEA Software Institute, Spain
GERMÁN ANDRÉS DELBIANCO, IRIF - Université Paris Diderot, France
IGNACIO FÁBREGAS, IMDEA Software Institute, Spain

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.

1 INTRODUCTION

The main problem when formally reasoning about concurrent data structures is achieving compositionality of proofs: how to ensure that methods of a data structure, once verified, can be used in a larger context without re-verification. There exist many prominent solutions to the problem, such as linearizability [Herlihy and Wing 1990], or more generally contextual refinement [Filipovic et al. 2010a; Liang and Feng 2018; Liang et al. 2014]. Most recently, Concurrent Separation Logic (CSL) [Brookes 2007; O’Hearn 2007] 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 [da Rocha Pinto et al. 2014; Dinsdale-Young et al. 2010; Jung et al. 2015; Nanevski et al. 2014; Svendsen and Birkedal 2014; Svendsen et al. 2013]—termed resources [Hoare 1972; O’Hearn 2007; Owicki and Gries 1976] 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 [1976] and the rely-guarantee method [Jones 1983], to facilitate modular reasoning in the presence of fine-grained interference. Some approaches [Frumin et al. 2018; Turon et al. 2013] 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, based on a notion of morphism-specific simulations, that uniformly handles many different functionalities that morphisms can encode. The upshot is conceptually simple foundations for separation logics for fine-grained concurrency.

1.1 Resource morphisms

Resource morphisms arise as a natural extension of the existing work on concurrent separation logics [da Rocha Pinto et al. 2014; Gratzer et al. 2019; Jacobs and Piessens 2011; Jung et al. 2018, 2015;
Svendsen and Birkedal 2014; Svendsen et al. 2013], 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 [da Rocha Pinto et al. 2014] 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 $\alpha$ 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 [Jacobs and Piessens 2011; Jung et al. 2015; Svendsen and Birkedal 2014; Svendsen et al. 2013] is to parametrize lock by $\alpha$, a priori, thus modifying it into:

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

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 $\alpha$ out of the spec of $\alpha$. We discuss such approaches further in Section 6.

In contrast, our approach employs morphisms to re-interpret the program a posteriori, as follows. We treat a resource as part of the program’s specification; thus it will appear as (part of) the program’s type. Execution of a program “inhabiting” a resource is a path through the resource’s graph. For example, lock can be typed by the resource Spin from Figure 1(a), and it describes paths through Spin consisting of several idle transitions $\text{id}_\text{tr}$, corresponding to unsuccessful CAS’s, followed by a locking transition $\text{lock}_\text{tr}$. A morphism can re-interpret $\text{lock}_\text{tr}$, e.g., to execute $\alpha$ in addition to locking, thus achieving the desired simultaneity.

Now, given resources $V$ and $W$, a resource morphism $f : V \rightarrow W$ consists of two partial functions $f_\Sigma$ and $f_\Delta$: $f_\Sigma$ takes a state in $W$ and produces a state, if defined, in $V$ (note the contravariance), and $f_\Delta$ takes a state in $W$ and transition in $V$ and produces a transition, if defined, in $W$. Combined, $f_\Sigma$ and $f_\Delta$ act on a program $e$ inhabiting $V$ to produce a program inhabiting $W$, using the following process.

Referring to 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 $f$ 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 $e$.

---

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 $\alpha$ is ghost code, the granularity of the program, and its behavior on real (i.e., non-ghost) state, remain unchanged.
The process determines a program in \( W \) that we denote \( \text{morph } f \ e \). Morph is a program constructor, and a form of function application of \( f \) to \( e \). As Section 2 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 \( \alpha \).

### 1.2 Simulations

To reason about \( \text{morph } f \ e \), we introduce the notion of morphism-specific simulations. An \( f \)-simulation is a predicate \( I \) over \( W \)-states satisfying, among other conditions listed in Section 3, the key technical property that the diagram 1(b) commutes, if \( I s_w \). Simulations provide a way to reason about \( \text{morph } f \ e \) compositionally out of \( e \)'s specification, with the following single inference rule, which is our second contribution:

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

Here \( f^P s_w \) is defined as \( 3s_w s_{Iw} = f_\Sigma s_w \land P s_v \). 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 \( [\text{O’Hearn et al. } 2001] \) of separation logic, a program \( e \) that is ascribed a Hoare triple isn’t stuck; thus there exists a transition \( t_v \) 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 \( s_v'' \), derives the postcondition above. First, \( Q s_v'' \) 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 \( I s_w'' \). We note that the diagram 1(b) doesn’t depend on the program \( e \) being verified, but only depends on the morphism \( f \) and the resources \( V \) and \( W \). Hence, the derived spec of \( \text{morph } f \ e \) depends only on the spec of \( e \), but not \( e \)'s code.

The paper can thus be seen as introducing simulations into separation logic in a simple\(^3\), 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,

\(^3\)Our logic has nine rules, each addressing an orthogonal linguistic feature.
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 formalize the development in Coq, using Coq’s predicates over states as assertions\(^4\), building on the previous code base of Fine-grained Concurrent Separation Logic (FCSL) [Nanevski et al. 2014]. The sources are available as a supplement [Nanevski et al. 2019]\(^5\).

2 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 [Herlihy and Wing 1990], and record the operations on $r$ in the linear sequence in which they occurred. We do so in Hoare triples, but in a thread-local way, i.e. from the point of view of the specified thread, which we refer to as “us” [Ley-Wild and Nanevski 2013]. Specifically, a program state $s$ contains a ghost component that we project as $\tau_s$, and which keeps “our” history of lock operations. Dually, the projection $\tau_o s$ keeps the collective history of all “other” (i.e. environment) threads. Each thread has these two components in scope, but they may have different values in different threads. We refer to $\tau_s$ and $\tau_o$ as self and other histories, respectively [Nanevski et al. 2014; Sergey et al. 2015], and denote by $\hat{\tau} s$ the union history $\tau_s \bullet \tau_o s$.

$$\begin{align*}
\text{lock} : [h, k]. \{ \lambda s. \tau_s s = h \land k \leq \text{last\_stamp}(\hat{\tau} s) \} \\
\{ \lambda s. \exists t. \tau_s s = h \land t \Rightarrow L \land k < t \} @\text{Spin}
\end{align*}$$

$$\begin{align*}
\text{unlock} : [h, k]. \{ \lambda s. \tau_s s = h \land k \leq \text{last\_stamp}(\hat{\tau} s) \} \\
\{ \lambda s. \exists t. \tau_s s = h \land t \Rightarrow U \land k < t \lor \\
\tau_s s = h \land \hat{\tau} s t = U \land k \leq t \} @\text{Spin}
\end{align*}$$

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 $\tau_s s$ equals $2 \Rightarrow U \land 7 \Rightarrow L \land 9 \Rightarrow L$, 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 unlocked 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 $\tau_s s$ equal to $h^0$ in the precondition, which is increased in the postcondition to log a locking event at time $t$. The conjunction $k < t$ in the

\(^4\)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 [Treiber 1986], flat combiner [Hendler et al. 2010], concurrent allocator, spanning tree of a graph, ticket [Mellor-Crummeay and Scott 1991] and readers-writers [Courtois et al. 1971] locks.

\(^6\)As customary in Hoare logic, $h$ and $k$ are logical variables, used to relate the pre and post-state. They are universally quantified, scoping over pre and post-condition, and the syntax $[\cdots]$ makes the binding explicit.
postcondition claims that \( t \) is fresh, because it’s larger than any \( k \) generated prior to the call (as \( k \leq \text{last\_stamp}(\hat{r}s) \) is a conjunct in the precondition, and \( k \) is universally quantified outside of the pre- and postcondition). The natural numbers ordering on timestamps gives the linear sequence in which the events logged in \( \hat{r}s \) occurred. Notice that the spec is stable, i.e., invariant under interference. Intuitively, other threads can’t modify the \( \tau s \) field, as it’s private to “us”. They can log new events into \( \tau s \), which features in the comparison \( k \leq \text{last\_stamp}(\hat{r}s) \), but this only increases the right-hand side of the comparison and doesn’t invalidate it. Similarly, unlock starts with history \( h \), which is either increased to log a fresh unlocking event at time \( t \), or remains unchanged if the unlocking fails because unlock encounters \( r \) already freed at time \( t (\hat{r}s t = U) \). The conjunct \( k \leq t \) captures that another thread may have freed \( r \) after the invocation of unlock \( (k < t) \), or that we invoked unlock with \( r \) already freed \( (k = t) \).

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 4, we will show how to apply morphisms to lock and unlock, to derive exclusive locking as in CSL [O’Hearn 2007], 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 [Nanevski et al. 2019], is to morph lock and unlock into non-exclusive locking protocols, such as readers-writers locks [Bornat et al. 2005; Courtois et al. 1971], where the first reader acquires the lock to allow read-only access to all reader threads, and
no access to writer threads. The last reader unlocks, and could be a different thread from the thread that locked. Thus, the specs, code, and proofs of lock and unlock can be morphed in different ways, fostering reuse.

Resources. Both specs are annotated by the resource Spin (defined in the left half of Figure 2). The state space of Spin makes explicit the assumptions about the components, i.e., that the histories are disjoint ( \( \tau_s s \perp \tau_o s \) ), that the entries in \( \hat{r} s \) alternate between L and U, and that \( r \) isn’t a null pointer. The erasure "s" shows how the state \( s \) maps to a heap once the ghost components are removed. In the case of Spin, the heap will contain only the pointer \( r \), set according to the total history. The transitions lock_tr and unlock_tr describe the allowed atomic modifications to the state\(^7\). For example, lock_tr adds a fresh L entry to \( \tau_s s \). When considered on the erased state, this corresponds to setting \( r \) to true. Thus, informally, lock and unlock can be implemented as follows, with ghost code in gray as before:

\[
\text{lock} = \text{do } (x \leftarrow \text{CAS}(r, \text{false}, \text{true});
\text{if } x \text{ then } \tau_s s := \tau_s s \bullet \text{fresh} (\hat{r} s) \Rightarrow L);
\text{while } \sim x\]

\[
\text{unlock} = (x \leftarrow !r; \ r := \text{false};
\text{if } x \text{ then } \tau_s s := \tau_s s \bullet \text{fresh} (\hat{r} s) \Rightarrow U)
\]

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 \( \kappa_s s \) of resource Counter (right half of Figure 2). We desire something like \( (\text{lock}; \ \kappa_s s := \kappa_s s + n) \) 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 lock_tr \( \bowtie \text{incr_tr} n \) and unlock_tr \( \bowtie \text{id_tr} \). Operator \( \bowtie \) (pronounced "couple") indicates that the operand transitions are executed simultaneously on their respective state halves. It’s defined as follows, where \( s \backslash 1 \) and \( s \backslash 2 \) project state \( s \) to its Spin and Counter components, respectively:

\[
(t_1 \bowtie t_2) s s' \equiv t_1 (s \backslash 1) (s' \backslash 1) \land t_2 (s \backslash 2) (s' \backslash 2)
\]

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

\[
\begin{align*}
(f n)_L s &= s \backslash 1 \\
(f n)_A s \text{ lock_tr} &= \text{lock_tr} \bowtie \text{incr_tr} n \\
(f n)_A s \text{ unlock_tr} &= \text{unlock_tr} \bowtie \text{id_tr}
\end{align*}
\]

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 \( (f n)_A \) on-the-fly. We can now achieve the desired a posteriori re-interpretation of lock as

\[
\text{morph } (f n) \text{ lock}
\]

---

\(^7\)By default, from now on, we elide the idle transition id_tr from diagrams.

\(^8\)The formal definition of tensoring is presented in Appendix A.
We start with the precondition because lock and $L$ morph for different instances of different execution of the morphed program, and into the post-state. The proof can use simulation as indicated. The simulation holds of the pre-state, and remains invariant throughout the morph to snapshot "our" history. In each application of annotates the invocations of the Spin program

$$
\text{lock}_\text{tr} \iff \left( f \cdot 1 \right)_\Delta \ s \ \text{lock}_\text{tr} \\
\left( f \cdot 1 \right)_\Sigma \ I_1 \ s
$$

Fig. 3. Diagram 1(b) specified for the case of $(f \cdot 1)_\Delta \ s \ \text{lock}_\text{tr} = \text{lock}_\text{tr} \triangleright \text{incr}_\text{tr}$ 1 and lock_tr s s’ $\equiv \neg \sigma_0 (\hat{r} \ s) \land \tau_s (s') = \tau_s \ s \ \cdot \ \text{fresh} (\hat{r} \ s) \Rightarrow L$.

to obtain a program in the SC resource, which executes lock_tr $\triangleright \text{incr}_\text{tr}$ n whenever lock executes lock_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

$$\text{morph} \ (f \cdot 1) \ \text{lock}; \quad \text{morph} \ (f \cdot 42) \ \text{unlock}; \quad \text{morph} \ (f \cdot 2) \ \text{lock}$$

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

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

We start with the precondition $\kappa_s (s \cdot 2) = n$, 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 1, 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 $f \cdot 1$, the simulation $I_1$ states that the counter kappa increments by the number of fresh L-entries in the history ($\#_L h$ is the number of L-entries in $h$). For the morphism $f \cdot 2$, $I_2$ says that $\kappa_s$ 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 3, but intuitively, each of $I_1$, $I_2$ and $I_{42}$ is a simulation for its respective morphism, because it satisfies the diagram 1(b) independently of the Spin transition $t_v$ chosen by the morphed program. For example, as Diagram 3 captures, $I_1$ is preserved by $t_v = \text{lock}_\text{tr}$, because lock_tr adds an L entry to $\tau_s$, and $(f \cdot 1)_\Delta \ s \ \text{lock}_\text{tr} = \text{lock}_\text{tr} \triangleright \text{incr}_\text{tr}$ 1 increments $\kappa_s$ by 1.
But, $I_1$ is also preserved by unlock_tr, as unlock_tr doesn’t add L entries, and $(f \ 1)_s \ s \ unlock_tr = unlock_tr \Rightarrow id_tr$ doesn’t change $\kappa_s$.

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 $I_3$ enables computing the end-value of $\kappa_s$ from the end-value of $\tau_s$, 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.

The rest of the paper proceeds as follows. We develop the concepts of resources, morphisms and simulations formally in Section 3, which is the technical contribution of the paper. We illustrate the use of these concepts by deriving CSL-style exclusive locking out of Spin in Section 4 (another Spin-based derivation, that of readers-writers locks, is in the Coq code). Finally, we show how generalization to morphism and simulation families applies to reasoning about allocation of new resources in Section 5.

### 3 DEFINITIONS OF THE FORMAL STRUCTURES

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

#### 3.1 States

As we can see 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, $M$ and $T$. The type $M$ classifies the self and other components, and the type $T$ classifies the joint (aka., shared) state. Thus, a state $s = (s_s, s_o)$ is an $(M, T)$-state iff $s_s, s_o \in M$, $s_o \in T$. 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. We use $s_s(s)$, $s_o(s)$ and $s_o(o)$ as generic projections out of $s$, but introduce labels for readability; e.g., $\tau_s/\tau_o$ and $\kappa_s/\kappa_o$ for $s_s/o_s$ 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 $s = (s_s(a), a_j, a_o)$, then the state of $\theta_2$ must be the transpose $s^T = (s_o, s_s(a), a_j)$. Both states represent the same configuration, with $s$ being the view of $\theta_1$ and $s^T$ that of $\theta_2$. The transposed states have swapped self and other components, thus capturing 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 $x \bullet y$ is defined.

Imagine three threads $\theta_1$, $\theta_2$ and $\theta_3$ running in parallel. Their respective states must have the forms $s_1 = (a_1, a_j, a_2 \bullet a_3)$, $s_2 = (a_2, a_j, a_3 \bullet a_1)$ and $s_3 = (a_3, a_j, a_1 \bullet a_2)$, 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 $s = (a_1 \bullet a_2, a_j, a_3)$, 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 $s_2$.

---

Section 4 will show an example with a non-trivial $T$. 

8 Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas.
We will similarly often move values back-and-forth between \( a_s \) and \( a_o \). To facilitate such surgery, we introduce the following operations on state, and notions of invariance under a change of viewing thread.

**Notation 3.1.** Let \( p \in M \) and \( s = (a_s, a_j, a_o) \) be an \((M, T)\)-state. The self-framing of \( s \) with the frame \( p \) is the state \( s \prec p = (a_s \cdot p, a_j, a_o) \). Dually, the other-framing of \( s \) with \( p \) is the state \( s \succ p = (a_s, a_j, p \cdot a_o) \).

We next define the notions of global predicates and functions. The intuition behind them is to describe properties and values that are invariant to all threads over a data structure.

**Definition 3.2.** A predicate \( P \) on \((M, T)\)-states is *global* if \( P(s \succ p) \iff P(s \prec p) \) for every \((M, T)\)-state \( s \) and \( p \in M \) such that \( a_s(s) \perp p \perp a_o(s) \). A (partial) function \( f \) on states is global if \( f(s \succ p) = f(s \prec p) \) under the same conditions. Given a global \( P \), the predicate \( Q \) is \( P\)-*global*, if \( P(s \prec p) \rightarrow Q(s \succ p) \iff Q(s \prec p) \), and similarly for functions.

Examples of global predicates from Section 2, are \( P_1 \) s = alternate \((\hat{t} s)\) used in Figure 2 to characterize Spin histories. \( P \) and \( Q \) are both defined in terms of \( \hat{t} s = t_o s \cdot \tau_o s \); \( Q \) directly so, and \( P \) because \( \tau_s s \perp \tau_o s \iff \tau_s s \cdot \tau_o s \) is itself defined. In other words, both \( P \) and \( Q \) express a property of the collective history of all threads operating over Spin, taken together. Clearly, the value of this history is invariant across all the threads, and therefore, so are \( P \) and \( Q \). Specifically, they are invariant under self and other-framings, as these shuffle timestamps between \( \tau_s \) and \( \tau_o \), but don’t alter the total. In fact, \( \hat{t} \) itself is global.

### 3.2 Resources

Because global predicates describe invariant properties, we next use them to represent state spaces of a resource.

**Definition 3.3.** An \((M, T)\) **state space** is a pair \( \Sigma = (P, \Gamma^{-\gamma}) \), where \( P \) is a global predicate on \((M, T)\)-state and \( \Gamma^{-\gamma} \) is a partial \( P\)-global function from \((M, T)\)-states into heaps, called *erasure*, such that for every state \( s \), \( Ps \) implies \( a_s(s) \perp a_o(s) \) and \( \Gamma^{-\gamma}s \) is defined. We write \( s \in \Sigma \) to mean \( Ps \).

For example, the state space of Spin in Figure 2 includes 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 \( s = s_1 * s_2 \) to predicates, and define the separation logic conjunction \( P * Q \), which will be used in the inference rules in Section 3.5.

\[
(P * Q)s \equiv \exists s_1, s_2. s = s_1 * s_2 \land Ps_1 \land Qs_2
\]

**Definition 3.4.** A **transition** \( t \) over \((M, T)\) state space \( \Sigma \) is a binary relation on \( \Sigma \) states, such that:

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

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

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 \( \text{lock}_\text{tr} \) in Figure 2 uses \( \tau_o s \) to compute a fresh timestamp. To explain transition locality (3), let \( \theta_i \) be a thread in the state \( s \succ p \), whose sibling \( \theta_o \) has self-component \( p \). Their parent \( \theta \) is thus in the state \( s \prec p \). If \( \theta_i \) performs a transition \( t(s \succ p) \), then by (3), the move can be seen as a transition of \( \theta \) in the state \( s \prec p \). Intuitively, the parent is “held responsible” for the transitions of its children.
Transition locality requires that the frame $p$ must be available in $a_0$, from which it is removed to be framed onto $a_1$. 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 $\tau_0 s$. In particular, a transfer of $p$ preserves the value of global functions, such as, e.g. $\hat{r} s$. Transition locality thus influences how we frame specs that contain global functions and properties. For example, we can give lock (Section 2) the following spec, which is small [O’Hearn et al. 2001] wrt. the history $\tau_s \mathcal{S}$.

\[
\text{lock : } [k]. \{ \lambda s. \tau_s s = 0 \land k \leq \text{last} \text{-} \text{stamp} (\tau_0 \mathcal{S}) \}
\{ \lambda s. \tau_s s = t \Rightarrow L \land k < t \} @ \text{Spin}
\]

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

\[
\text{lock : } [h, k]. \{ \lambda s. \tau_s s = h \land k \leq \text{last} \text{-} \text{stamp} (\hat{r} \mathcal{S}) \}
\{ \lambda s. \tau_s s = h \land t \Rightarrow L \land k < t \} @ \text{Spin}
\]

Notice how the unframed version applies last_stamp to $\tau_0 \mathcal{S}$, whereas the framed version uses $\hat{r} s = \tau_s \mathcal{S} \mathcal{S} \tau_0 \mathcal{S}$, to compensate for the fact that framing added $h$ to $\tau_s \mathcal{S}$ and thus removed it from $\tau_0 \mathcal{S}$. 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 $\tau_0 \mathcal{S}$ gives us a local way to work with global properties.

**Definition 3.5.** A $\Sigma$-transition $t$ is **footprint preserving** if $t s s’$ implies that $\mathcal{S} s’$ and $\mathcal{S} s’$ 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 [Nanevski et al. 2019]. 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 beensored with an allocator resource to exchange pointers through ownership transfer [Filipovic et al. 2010b; Nanevski et al. 2014] via external transitions. We elide further discussion, but refer to the Coq files for the implementation of an allocator resource and example programs that use it.

**Definition 3.6.** A **resource** is a tuple $V = (M, T, \Sigma, \Delta)$, where $\Sigma$ is an $(M, T)$ state space, and $\Delta$ a set of footprint preserving $\Sigma$ transitions. We refer to $V$’s components as projections, e.g. $\Sigma(V)$ for the state space, $\Delta(V)$ for the transitions, $M(V)$ for the PCM, etc. A state $s$ is $V$-state iff $s \in \Sigma(V)$.

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

**Definition 3.7.** An **action** of type $A$ in a resource $V$ is a partial function $a : \Sigma(V) \rightarrow \Delta(V) \times A$, mapping input state to output transition and value, which is **local**, in the sense that it is invariant under framing. Formally, if $a (s \triangleright p) = (t, v)$ then $a (s \triangleleft p) = (t, v)$; that is, if $a$ is performed by a child thread, it behaves the same when viewed by the parent.

The **effect** of $a$ is the partial function $[a] : \Sigma(V) \rightarrow \Sigma(V) \times A$ mapping input state to output state and value, defined as $[a] s = (s’, v)$ iff $\exists t. a s = (t, v) \land t s s’$. 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 2, as the following action of type bool:

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

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

We say that a erases to an atomic read-modify-write (RMW) command c [Herlihy and Shavit 2008], if \([a] s = (s', v)\), then \( c \circ s^\gamma = (\tau s'^\gamma, v) \). One may check that trylock\_act erases to CAS(r, false, true), as expected\(^{10}\). Similarly,

\[
\text{unlock\_act } s = \begin{cases} 
\text{unlock\_tr, ()} & \text{if } \omega(\hat{\tau} s) \\
\text{id\_tr, ()} & \text{otherwise}
\end{cases}
\]

is an action of unit type, which erases to \( r := \bot \).

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 1.

### 3.3 Morphisms

Definition 3.8. A morphism \( f : V \rightarrow W \) consists of partial functions \( f_\Sigma : \Sigma(W) \rightarrow \Sigma(V) \) (note the contravariance), and \( f_\Lambda : \Delta(W) \rightarrow \Delta(V) \), such that:

1. (locality of \( f_\Sigma \)) there exists a function \( \phi : M(W) \rightarrow M(V) \) such that if \( f_\Sigma(s_w \triangleright p) = s_v \), then there exists \( s_v' \) such that \( s_v = s_v' \triangleright \phi(p) \), and \( f_\Sigma(s_w \triangleleft p) = s_v' \triangleleft \phi(p) \).
2. (locality of \( f_\Lambda \)) if \( f_\Lambda(s_w \triangleright p)(t_v) = t_w \), then \( f_\Lambda(s_w \triangleleft p)(t_v) = t_w \).
3. (other-fixity) if \( a_o(s_w) = a_o(s_w') \) and \( f_\Sigma(s_w), f_\Sigma(s_w') \) exist, then \( a_o(f_\Sigma(s_w)) = a_o(f_\Sigma(s_w')) \).

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

The properties (1) and (2) of Definition 3.8 provide basic technical conditions for this process to be invariant under framing. Property (1) is a form of “simulation of framing”, i.e., a frame \( p \) in \( W \) can be matched with a frame \( \phi(p) \) in \( V \). Thus, framing a morphed program can be viewed as framing the original program. Property (2) says that framing doesn’t change the transition that \( f_\Lambda \) produces; thus it doesn’t influence the behavior of morphed programs. The property (3) restricts the choice of \( s_v' \) in (1) so that that \( a_o(s_v') \) is uniquely determined by \( a_o(s_w) \), much as how \( \phi(p) \) in (1) is uniquely determined by \( p \). This is a technical condition which we required to prove the soundness of the frame rule.

Example. Properties (1)-(3) are all satisfied by the morphism family \( f n : \text{Spin} \rightarrow \text{SC} \) from Section 2. Indeed, \( M(\text{SC}) = M(\text{Spin}) \times M(\text{Counter}) = \text{histories} \times \mathbb{N} \). Thus, a frame in 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 \( f n_\Lambda \) doesn’t depend on the state argument, hence framing this state doesn’t change the output. Finally, in (3), the values \( a_o(s_w) \) and \( a_o(s_w') \) are also pairs of a history and a nat. If the pairs are equal, then their history components are equal too, deriving (3).

\(^{10}\) All the actions we use in this paper and in the Coq code erases to some RMW command. However, we proved this only by hand, as the Coq implementation doesn’t currently issue proof obligations to check this.
### 3.4 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 $s_U = f_\Sigma s_w$, whose computation is the first step of morphing, needn’t exist. Even if $s_U$ does exist, and the original program takes the transition $t_U$ in $s_U$, then $t_w = f_\Delta s_w t_U$ 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 3.9.** 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 $f_\Sigma (s_w) = s_U$, and $t_U s_U s'_U$, then there exist $t_w = f_\Delta s_w t_U$ and $s'_w$ such that $I s'_w$ and $f_\Sigma (s'_w) = s'_U$, and $t_w s_w s'_w$.
2. if $I s_w$, and $f_\Sigma (s_w)$ exists, and $s_w \rightarrow^* s'_w$, then $I s'_w$, and $f_\Delta (s'_w)$ exists, and $f_\Sigma (s_w) \rightarrow^* f_\Delta (s'_w)$.

Here, the relation $s \rightarrow s'$ denotes that $s$ other-steps by $W$ to $s'$, i.e., that there exists a transition $t \in \Delta (W)$ such that $t s t^* s'$. The relation $\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$, allowing for an arbitrary number of steps.

Property (1) says that $W$ simulates $V$ on states satisfying $I$. Property (2) states the simulation in the opposite direction, i.e., of $W$ by $V$, but allowing many other-steps to match many other-steps. Intuitively, (2) ensures that interference in $W$ may be viewed as interference in $V$, so that stable Hoare triples in $V$ can be transformed into stable Hoare triples in $W$, which is required for the soundness of morph rule. Condition (1) has already been shown in diagram 1(b); we repeat it in Figure 4, together with a diagram for condition (2).

Morphisms possess the following basic categorical structure.

**Definition 3.10.** Composition $g \circ f$ of morphisms $f : U \rightarrow V$ and $g : V \rightarrow W$ is a morphism from $U$ to $W$ defined by:

\[
(g \circ f)_\Sigma s_w = f_\Sigma (g_\Sigma s_w) \\
(g \circ f)_\Delta s_w t_U = g_\Delta s_w (f_\Delta (g_\Sigma s_w) t_U)
\]

**Identity** morphism id : $V \rightarrow V$ is defined by id$_\Sigma$ $s = s$ and id$_\Delta$ $s t = t$. Composition is associative, with unit id, where two morphisms are equal if their $\Sigma$ and $\Delta$ components are equal as partial functions.

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 $V$. 

![Fig. 4. Conditions for $I$ to be an $f$-simulation.](image-url)
3.5 Inference rules

We present the system using the Calculus of Inductive Constructions (CiC) as an environment logic, hence as a shallow embedding in Coq. We inherit from CiC the useful concepts of higher-order functions and substitution principles, and only present the notions specific to Hoare logic\(^{11}\).

We differentiate between two different notions of program types: \(STVA\) 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 2). The second type is a subset of \(STVA\), 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 \(vrf\) which takes a program \(e : STVA\), 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 types are then defined in terms of \(vrf\), as follows.

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

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

\[
fix : (T \rightarrow T) \rightarrow T
\]

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 fix, are the only Hoare-related rules of the system. In the rules we assume that \(e : STVA, e_i : STVA_i, a\) is a \(V\)-action, \(f : V \rightarrow W\) is a morphism, \(I\) is an \(f\)-simulation, \(J\) is a \(V\)-invariant, and \(s \in \Sigma(V)\).

\[
\begin{align*}
vrf\_post & : (\forall v. J s \rightarrow Q_1 v s \rightarrow Q_2 v s) \rightarrow J s \rightarrow vrf\ e\ Q_1 s \rightarrow vrf\ e\ Q_2 s \\
vrf\_ret & : (Q v)^* s \rightarrow vrf\ (\text{ret}\ v)\ Q s \\
vrf\_bnd & : vrf\ (\lambda v. vrf\ (e_2\ x)\ Q)\ s \rightarrow vrf\ (x \leftarrow e_1; (e_2\ x))\ Q s \\
vrf\_par & : (\lambda e_1\ Q_1) \ast (\lambda e_2\ Q_2)\ s \rightarrow vrf\ (e_1\ ||\ e_2)\ (\lambda v : A_1 \times A_2, (Q_1\ v.1) \ast (Q_2\ v.2))\ s \\
vrf\_frame & : (vrf\ e\ Q_1)^* Q_2^* s \rightarrow vrf\ (\lambda v. (Q_1\ v)\ast Q_2)\ s \\
vrf\_act & : (\lambda s'. \exists s''. \exists' v. [a] s' = (s''', v) \land (Q v)^* s')^* s \rightarrow vrf\ (a)\ Q s \\
vrf\_morph & : f^*(vrf\ e\ Q)\ s \rightarrow I s \rightarrow vrf\ (\text{morph}\ f\ e)\ (\lambda v\ s'. f^*(Q\ v)\ s' \land I\ s')\ s
\end{align*}
\]

In English:

- The \(vrf\_post\) rule weakens the postcondition, similar to the well-known rule in Hoare logic.
  The rule allows assuming a \(V\)-invariant \(I\) when establishing a postcondition \(Q_2\) out of \(Q_1\).
  Thus, a resource invariant can be elided when deriving one spec from another, as it can be invoked by \(vrf\_post\) as needed.
- The \(vrf\_ret\) rule applies to an idle program returning \(v\). When we want an idle program that returns no value, we simply take \(v\) to be of unit type. The rule explicitly \textit{stabilizes} the postcondition \(Q\) to allow for the state \(s\) to be changed by interference of other threads in between the invocation of the idle program and its termination. Here, stabilization of a

\(^{11}\text{Appendix D defines the denotational semantics, in CiC, for these notions, and states a theorem, proved in Coq, that the inference rules are sound wrt. the denotational semantics.}\)
We next illustrate a more involved application of morphisms and simulation: how to derive a
want to combine the
\[ Q^* (s) \equiv \forall s'. s \rightarrow^\nu s' \rightarrow Q(s') \].

\[ Q \] is stable if \( Q = Q^* \), and it is easy to see
that \( Q^* \) is stable for every \( Q \).

- The vrf_bnd rule is a Dijkstra-style rule for sequential composition.
- The vrf_par and vrf_frame rules are predicate transformer variants of the rules for parallel composition and framing from separation logic. The vrf_frame rule can be seen as an instance of vrf_par, where \( e_2 \) is taken to be the idle programs returning no value. Thus, \( Q_2 \) is explicitly stabilized in vrf_frame, to match the precondition of the vrf_ret rule for idle programs.
- The vrf_act rule says that \( Q \) holds after executing action \( a \) in state \( s \), if \( s \) obtains \( s' \) by interference of other threads, and then \( [a] s' \) returns the pair \( (s'', v) \) of output state and value. The latter satisfy the stabilization of \( Q \), to allow for interference on \( s'' \) after the termination of \( a \).
- Finally, the vrf_morph rule is a straightforward casting of the morph rule from Section 1 into a predicate transformers style.

4 EXCLUSIVE LOCKING VIA MORPHING AND THE NEED FOR PERMISSIONS

We next illustrate a more involved application of morphisms and simulation: how to derive a
resource and methods for exclusive locking, à la CSL [O’Hearn 2007], from a resource for general
locking (Section 2). An exclusive lock protects a shared heap, satisfying a user-supplied predicate \( R \)
(aka. resource invariant). Upon successful locking, the shared heap is transferred to the private
ownership of the locking thread, where it can be modified at will, potentially violating \( R \). Before
unlocking, the owning thread must re-establish \( R \) in its private heap, after which, the part of the
heap satisfying \( R \) is moved back to the shared status. The idea is captured by the following methods
and specs, which we name exlock and exunlock to differentiate from lock and unlock in Section 2.

\[
\text{exlock} : \{ \lambda s. \mu_5 s = \overline{\text{own}} \land \chi_5 s = \emptyset \}
\]
\[
\{ \lambda s. \mu_5 s = \text{own} \land R(\chi_5 s) \} \text{@CSL}
\]
\[
\text{exunlock} : \{ \lambda s. \mu_5 s = \text{own} \land R(\chi_5 s) \}
\]
\[
\{ \lambda s. \mu_5 s = \overline{\text{own}} \land \chi_5 s = \emptyset \} \text{@CSL}
\]

Here \( \mu_5 s \) is a ghost component of type \( O = \{ \text{own}, \overline{\text{own}} \} \), signifying whether “we” own the lock
or not, and \( \chi_5 s \) is “our” private heap. \( O \) has PCM structure with the join defined by
\( x \land \overline{\text{own}} = \overline{\text{own}} \land x = x \), so that \( \overline{\text{own}} \) is the unit of the operation. We leave \( \text{own} \land \text{own} \) 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 5, where the CSLX resource contains the state components
and transitions describing the functionality needed for heap transfers. In particular, \( \sigma_s s \in O \) keeps
track of whether we own the lock or not, \( \sigma_5 \) and \( \sigma_\_ \) are the private and shared heap respectively, and
the transitions open_tr and close_tr move the heap from shared to private and back, respectively.
We want to combine the Spin and CSLX resources as shown in the figure, by combining their state spaces,
and coupling open_tr with lock_tr, and close_tr with unlock_tr, so that the transitions execute
simultaneously. This will give us an intermediate resource CSL’, and a morphism \( f : \text{Spin} \rightarrow \text{CSL’} \)
defined similarly as in Section 2:

\[
f_{\#} s = s \backslash 1
\]
\[
f_{\#} s \text{ lock_tr} = \text{lock_tr} \Leftrightarrow \text{open_tr}
\]
\[
f_{\#} s \text{ unlock_tr} = \text{unlock_tr} \Leftrightarrow \text{close_tr}
\]
In general, the design of resource’s permissions obviously and essentially influences how that resource composes with others. Some systems, such as CAP [Dinsdale-Young et al. 2010] and iCAP [Svendsen and Birkedal 2014], although they don’t consider morphisms and simulations, by default provide a permission for each transition of a resource. In our example, that would correspond to also having a permission for lock_tr. For simplicity, we elide such generality, and consider only the permission to unlock, which suffices to illustrate morphisms and simulations.
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\textsuperscript{13}.

\[ \text{lock’} : [h, k] \text{.} \begin{cases} \lambda s. \tau_s s = h \wedge k \leq \text{last\_stamp} (\hat{\tau} s) \wedge \pi_s s = 0 \\ \lambda s. \exists \tau. \tau_s s = h \bullet t \Rightarrow L \wedge k < t \wedge \pi_s s = 1 \end{cases} @\text{Spin} \]

\[ \text{unlock’} : [h, k] \text{.} \begin{cases} \lambda s. \tau_s s = h \wedge k \leq \text{last\_stamp} (\hat{\tau} s) \wedge \pi_s s = 1 \\ \lambda s. \exists \tau. \tau_s s = h \bullet t \Rightarrow U \wedge k < t \wedge \pi_s s = 0 \wedge \tau_s s = h \wedge \hat{\tau} s t = U \wedge k \leq t \wedge \pi_s s = 1 \end{cases} @\text{Spin} \]

Let us now consider the combination CSL’ of Spin and CSLX as defined in Figure 5. 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

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

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

as a basic coherence property\textsuperscript{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:

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

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

(3) where \(h \perp \omega k\) is defined as

\[ (\text{last\_op } h = L \rightarrow \text{last\_stamp } k < \text{last\_stamp } (h)) \wedge (\text{last\_op } k = L \rightarrow \text{last\_stamp } h < \text{last\_stamp } (k)) \wedge h \perp k \]

to say that if \(h\) (resp. \(k\)) indicates that a thread holds the lock, then another thread couldn’t have proceeded to add logs to its own history \(k\) (resp. \(h\)), and unlock itself\textsuperscript{15}.

It is now easy to see that \(\text{Inv} = (1) \wedge (2) \wedge (3)\) is an invariant of CSL’. The critical point is that (3) is preserved by the \(t = \text{unlock\_tr} \Rightarrow \text{close\_tr}\) transition. Indeed, if in state \(s \in \text{Inv}\), we transition by \(t\), then we add a fresh unlocking entry to the ending state \(s’\), thus making \(\text{last\_stamp } (\tau_s s’) > \text{last\_stamp } (\tau_o s’).\) For (3) to be preserved, it must then be \(\text{last\_op } (\tau_o s’) = U\), 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 \(s \in \text{Inv}\), if we are the thread that last 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 \(\text{Inv}\) is invariant, we can construct a resource CSL out of CSL’, where \(\text{Inv}\) is imposed as an additional property of the underlying PCM and state space of CSL’. Indeed, our theory ensures that the set \(\Sigma (\text{CSL’}) \cap \text{Inv}\) can be made a global predicate, and thus be used as a state space of a new resource CSL. By Definition 3.2, locality depends on the underlying PCM, hence the construction

\[ \text{Remark (3) restricts only the last timestamp in } h \text{ and } k, \text{ not all timestamps hereditarily. This suffices for our proof.} \]

\[ \text{The specs for lock’ and unlock’ are small wrt. } \pi_s s, \text{ but large wrt. } \tau_s s. \]

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

\[ \text{Requirement (3) restricts only the last timestamp in } h \text{ and } k, \text{ not all timestamps hereditarily. This suffices for our proof.} \]
involves restricting the PCM of CSL’ by \( \text{Inv} \). The mathematical underpinnings of such restrictions involve developing the notions of sub-PCMs, PCM morphisms and compatibility relations, which we defer to Appendix B. Here, it suffices to say that the construction leads to the situation summarized by the following diagram:

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

where morphism \( i \) is defined by \( i_2. s = s \) and \( i_3. t = t \). Intuitively, CSL states are a subset of CSL’ states satisfying \( \text{Inv} \), and \( i_2 \) is the injection from \( \Sigma(\text{CSL}) \) to \( \Sigma(\text{CSL'}) \).

This gives us the CSL resource, but we still need to transform \( \text{lock'}/\text{unlock'} \) into \( \text{exlock}/\text{exunlock} \), respectively. We thus introduce the following property on CSL states:

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

Unlike \( \text{Inv} \), \( \text{Sim} \) 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, \( \text{Sim} \) is an \( (i \circ f) \)-simulation, as it satisfies the commuting diagrams from Figure 4. 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, \( \text{Sim} \) describes the state of CSL immediately after locking, and immediately before unlocking, which suffices for the morphing of lock’ and unlock’. We only show the derivation for \( \text{exunlock} = \text{morph } (i \circ f) \text{ unlock' } \), and refer to the Coq code [Nanevski et al. 2019] for the derivation of exlock.

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

The key step is in line 6, where we must derive that the second disjunet 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 disjunet is false by reasoning about the histories \( \tau_s s \) and \( \tau_0 s \) in line 5. From \( \pi_s s = 1 \), it must be \( \omega(\tau_s s) = \text{true}, \) by the invariant \( \text{Inv} \) which holds throughout, as \( s \) is a CSL state. In particular, \( \text{last \_op}(\tau_s s) = L. \) By \( \text{Inv} \) again, \( \tau_s s \) and \( \tau_0 s \) are exclusive histories, so \( \text{last \_stamp}(\tau_0 s) < \text{last \_stamp}(\tau_s s) \), and it must be \( \text{last \_stamp}(\tau_s s) = \text{last \_stamp}(\hat{t} s) = k, \) because \( \text{last \_stamp}(\hat{t} s) \) is the maximum of \( \text{last \_stamp}(\tau_s s) \) and \( \text{last \_stamp}(\tau_0 s) \). But this contradicts that \( \hat{t} s \) contains entry \( U \) at \( t \geq k \). We now obtain the desired spec of exunlock from the beginning of the section, by taking \( \mu = \alpha \) and \( \chi = \sigma. \)

5 QUIESCENCE AND INDEXED MORPHISM FAMILIES

The examples in Sections 2 and 4 were examples of abstract atomicity [da Rocha Pinto et al. 2014], 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 \( f : V \to W \) introduces a type \( X \) of indices for \( f \). The state component \( f_\Sigma : X \to \Sigma (W) \to \Sigma (V) \) and the transition component \( f_\Delta : X \to \Sigma (V) \to \Delta (V) \to \Delta (W) \) now allow input \( X \), and satisfy a number of properties, listed in Appendix C, which reduce to Definition 3.8 when \( X = \text{unit} \). Similarly, \( f \)-simulations must be indexed too, to be predicates over \( X \) and \( \Sigma (W) \), satisfying a number of properties which reduce to Definition 3.9 when \( X = \text{unit} \).

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

\[
\text{morph } f \ x \ e : \{ \lambda \ell . \text{P} \} \ A \{ Q \} @V
\]

Here, \( ( f \ x)^\ast P s_w \) is defined as \( \exists s_w . \ s_w = f_\Sigma x s_w \wedge P s_w \).

To illustrate, consider a spec for a concurrent stack’s push method, similar to that of lock from Section 1:

\[
\text{push}(\nu) : \{ k \}. \{ \lambda s . \sigma_s s = \emptyset \wedge \tau_s s = \emptyset \wedge k \leq \text{last\_stamp}(\tau_o s) \}
\]

\[
\{ \lambda s . \sigma_s s = \emptyset \wedge \exists t . \upsilon_s . \tau_s s = t \implies (\upsilon_s \upsilon :: \upsilon s) \wedge k < t \} \text{@Stack}
\]

Here \( \tau_s \) and \( \tau_o \) are histories of stack’s operations. In the case of Spin, and \( \sigma_s \) is the thread-private heap. The spec says that push starts with \( \tau_s s = \emptyset \) (by framing, any history) and ends with \( \tau_s s = t \implies (\upsilon s \upsilon :: \upsilon s) \) to indicate that a push of \( \upsilon \) indeed occurred, and after all the timestamps from the pre-state. The Stack resource also has joint components \( \sigma_f \) (the heap storing the stack’s physical layout) and \( \sigma_j \) (abstract contents of the stack as a mathematical sequence). The push method first allocates a new node in \( \sigma_s \), then moves it to \( \sigma_f \) where it is enlinked to the top of the laid-out stack, after which push updates \( \tau_s \) and \( \tau_f \) 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 \( \sigma_f \) is laid out in \( \sigma_j \) (i.e., \( \forall s \in \Sigma (\text{Stack}) \). layout \( (\sigma_f s)(\sigma_j s) \)), and that the global history \( \hat{r} s \) has no timestamp gaps (i.e., \( \forall s \in \Sigma (\text{Stack}) \). dom \( (\hat{r} s) = \{ 1, \ldots, \text{last\_stamp}(\hat{r} s) \} \)).

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

\[
e : \{ \lambda s . \sigma_s s = \emptyset \wedge \tau_s s = \emptyset \}
\]

\[
\{ \lambda s . \sigma_s s = \emptyset \wedge \exists t_1 \upsilon s_1 \ t_2 \upsilon s_2 . \tau_s s = t_1 \implies (\upsilon s_1 a :: \upsilon s_1) \bullet t_2 \implies (\upsilon s_2 b :: \upsilon s_2) \} \text{@Stack}
\]

The specification reflects that \( e \) pushes \( a, b \), to change the stack contents from \( \upsilon s_1 \) to \( a :: \upsilon s_1 \) at time \( t_1 \), and from \( \upsilon s_2 \) to \( b :: \upsilon s_2 \) 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 \( \bullet \) is commutative, the order of \( t_1 \) and \( t_2 \) in the binding to \( \tau_s s \) 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., quiescently [Aspnes et al. 1994; Derrick et al. 2011; Jagadeesan and Riely 2014; Nanevski et al. 2014; Sergey et al. 2016]? For example, imagine a resource 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 [Herlihy and Shavit 2008][10]. A program working over 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

\[\text{footnote}{[10]}\text{We implemented such a resource Priv in our Coq files [Nanevski et al. 2019]. It corresponds to a separation logic with no ghost state.}\]
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:

\[
\{ \lambda s. \text{layout}(\sigma_s(s)) \} \{ \lambda s. \text{layout}[a, b](\sigma_s(s)) \lor \text{layout}[b, a](\sigma_s(s)) \} @ \text{Priv}
\]

The move from \text{Stack} to \text{Priv} thus essentially forgets the ghost state of histories, and the distinction in \text{Stack} between shared and private heaps. These components and distinctions are visible when in the scope of \text{Stack}, but hidden when in \text{Priv}\(^{17}\). We would like to obtain the \text{Priv} spec by applying the morph rule to the \text{Stack} spec of \( e \), with a morphism \( f : \text{Stack} \to \text{Priv} \) 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 \( s_{\text{Priv}} \), 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 \( f : \text{Stack} \to \text{Priv} \) a family over \( X = \text{hist} \), thereby passing to \( f_{\Sigma} \) the history \( x \) that should be added to \( s_{\text{Priv}} \) to produce an \( s_{\text{Stack}} \).

\[
f_{\Sigma} \times s_{\text{Priv}} = s_{\text{Stack}} \triangleq \sigma_s(s_{\text{Priv}}) = \sigma_s(s_{\text{Stack}}) \land \sigma_o(s_{\text{Priv}}) = \sigma_o(s_{\text{Stack}}) \land \tau_s(s_{\text{Stack}}) = x \land \tau_o(s_{\text{Stack}}) = \emptyset
\]

The first conjunct directly states that \text{Stack} is installed in \( \sigma_s(s_{\text{Priv}}) \) by making \( \sigma_s(s_{\text{Priv}}) \) be the join of the heaps \( \sigma_j(s_{\text{Stack}}) \) and \( \sigma_s(s_{\text{Stack}}) \). The second conjunct says that the heap \( \sigma_o(s_{\text{Priv}}) \) of the interfering threads is propagated to \( \sigma_o(s_{\text{Stack}}) \). The third conjunct captures that the history component of \( s_{\text{Stack}} \) is set to the index \( x \), as discussed immediately above. In the last conjunct, the \( \tau_o(s_{\text{Stack}}) \) history is declared \( \emptyset \), thus formalizing quiescence. We elide the definition of \( f_{\Delta} \); it suffices to know that it maps a Stack transition (relation over heap and ghost state of \text{Stack}) to its “erasure”, i.e. a relation over heaps of \text{Priv} representing a single-pointer operation such as read, write and CAS, but ignoring the ghost histories\(^{19}\). Finally, applying the new morph rule with \( f \) to the Stack specification of \( e \), with \( f \) being the always-true predicate on \text{Priv} states, and \( x = \emptyset \), gives us exactly the desired quiescent \text{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 \( \tau_s(s_{\text{Stack}}) = y = t_1 \mapsto (v_{s_1}, a :: v_{s_1}) \bullet t_2 \mapsto (v_{s_2}, b :: v_{s_2}) \), and \( \tau_o(s_{\text{Stack}}) = \emptyset \), and \( \sigma_s(s_{\text{Priv}}) = \sigma_j(s_{\text{Stack}}) \). From \( \Sigma(\text{Stack}) \), we know layout \( (a_j(s_{\text{Stack}}))(\sigma_j(s_{\text{Stack}})) \), and hence layout \( (a_j(s_{\text{Stack}}))(\sigma_j(s_{\text{Priv}})) \). Also from \( \Sigma(\text{Stack}) \), we know that \( \tilde{\tau}(s_{\text{Stack}}) \) has no timestamp gaps; thus \( \{t_1, t_2\} = \text{dom}(\tilde{\tau}(s_{\text{Stack}})) = \{1, 2\} \), i.e., \( t_1 \) and \( t_2 \) are the only, and consecutive, events. But then \( a_j(s_{\text{Stack}}) \) must be either \([a, b]\) or \([b, a]\), giving us the desired postcondition in \text{Priv}.

6 RELATED WORK

Coalgebra morphisms. State transition systems are mathematically representable as coalgebras [Jacobs 2016; Rutten 2000]; thus, resource morphisms are closely related to coalgebra morphisms. There are, however, differences between our definition of morphisms and the coalgebraic one, arising from our application to concurrent separation logic. For example, for us, given \( f : V \to W \), \( f_{\Sigma} \) is contravariant, but in the coalgebraic case, \( f \) is covariant on states [Hasuo et al. 2009; Rutten 2000]. A coalgebra morphism \( f \) doesn’t have the \( f_{\Delta} \) component, but requires that \( f \) preserves and

---

\(^{17}\)Such management of the scope of state is related to what the past work has dubbed anti-framing [Pottier 2008; Schwinghammer et al. 2013].

\(^{18}\)As we want to build \( s_{\text{Stack}} \) out of \( s_{\text{Priv}} \), we have to identify the part of \( \sigma_s(s_{\text{Priv}}) \) which we want to assign to \( \sigma_j(s_{\text{Stack}}) \). 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.
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 3.9 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 [2011] have originated an approach to abstract atomicity (later expanded in HOCAP [Svendsen et al. 2013], iCAP [Svendsen and Birkedal 2014], and Iris [Jung et al. 2018, 2015]), 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 2, the 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 2 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 [1990], 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 [Delbianco et al. 2017]. 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.

The TADA logic of da Rocha Pinto et al. [2014], introduces a new judgment form, $\langle P \rangle e \langle Q \rangle$, that captures that $e$ has a precondition $P$ and postcondition $Q$, but is also abstractly atomic in the following sense: $e$ and its concurrent environment maintain the validity of $P$, until at one point $e$ 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 [Hendler et al. 2010], where one thread executes work on behalf of others; helping is an idiom that TADA can’t currently express.

In Iris [Jung et al. 2015], one of the key features is that of invariants, whose definition is similar to our invariants. For us, however, invariants arise as a special case of $f$-simulations, when the morphism $f$ is the identity. Thus, the basis of reasoning in our logic are simulations, which are a standard concept in concurrency verification. In contrast, Iris focuses on the orthogonal notion of impredicative higher-order state. This leads to significant differences in the use of the two logics. For example, $f$-simulations enable us to use a single rule morph, which applies generically to all programs. In Iris, reasoning by invariants is supported by a number of rules (e.g. INV in loc.cit.), but many apply to atomic programs only. In fact, the restriction of such rules to atomic programs is used as a direct motivation for the parametrization approach. That said, our present model (Appendix D) isn’t impredicative, and it remains future work to introduce impredicativity into it.
We noted that resource morphisms go beyond abstract atomicity, as they apply to quiescence (Section 5), 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. [2018] 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.\( \text{id}_\text{tr} P = \lambda s\ s'. P \land s' = s. \) Disel doesn’t currently consider hooks where both operands are non-idle, which we used in the lock examples, or notions of morphism and simulation. On the other hand, we haven’t considered distribution so far.

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

There are several advantages of this approach over refinement and linearizability. First, a spec in the form of a Hoare triple with a resource is much simpler, and thus easier to establish than a spec in the form of another program. Indeed, resources aren’t programs; they are STSs and don’t admit programming constructs such as conditionals, loops, initial or local state, or function calls. A Hoare spec is also immediately useful in proofs, whereas with 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 [Cerone et al. 2014; Gotsman and Yang 2012], whereas for us (and other extensions of CSL [da Rocha Pinto et al. 2014; Dinsdale-Young et al. 2010; Jung et al. 2018, 2015; Liang and Feng 2013]), ownership transfer is directly inherited from separation logic. We also inherit from separation logic a way to dynamically nest parallel compositions of threads, whereas linearizability is typically considered on programs with a fixed, though arbitrary, number of threads. Finally, our specs of lock and unlock in Section 2 are actually very close to what one gets from linearizability, as they essentially establish a linear order between locking and unlocking events in the history PCM. However, as shown by Sergey et al. [2016], the PCM-based approach 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 [Nanevski et al. 2014] the novel notions of resource morphism, and significantly modifies the notion of resources. In FCSL, each concurrent resource is a finite map from labels (natural numbers) to sub-components. For example, using the concepts from Section 2, one could represent SC as a finite map \( l_1 \models \text{Spin} \uplus l_2 \models \text{Counter} \), where \( l_1 \) and \( l_2 \) are labels identifying Spin and Counter, respectively. This approach provides interesting equations on resources; for example, one can freely rearrange the finite map components by using commutativity and associativity of \( \uplus \). 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 2 into SC’ by commuting Spin and Counter throughout the construction, then SC and SC’ wouldn’t be equal resources, but they will be isomorphic, in that we could exhibit cancelling morphisms between the two. But this requires first having a notion of morphism, which is the technical contributions of this paper. FCSL supported quiescence by means of a dedicated and very complex inference rule, whereas Section 5 demonstrates that quiescence is merely an application of morphism families.

7 CONCLUSIONS

This paper develops novel notions of resource morphisms and associated simulations, as key mathematical concepts that underpin a separation logic for fine-grained concurrency. This is a natural development, as structures in mathematics are always associated with an appropriate notion of morphism.

A morphism from resource $V$ to resource $W$ exhibits a form of forward simulation [Abadi and Lamport 1991] 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 2, is specialized to a specific ownership discipline, e.g., exclusive locking in Section 4 (though we have also developed [Nanevski et al. 2019] an instantiation to readers-writers [Courtois et al. 1971] locking). Other applications include the managing of scope of ghost state in quiescent environments, as illustrated in Section 5, and restricting the state space of a resource with additional state and PCM invariants, as used in Section 4.

REFERENCES

Martin Abadi and Leslie Lamport. 1991. The existence of refinement mappings. Theoretical Computer Science (TCS) 82, 2 (1991), 253–284.

James Aspnes, Maurice Herlihy, and Nir Shavit. 1994. Counting Networks. J. ACM 41, 5 (1994), 1020–1048.

Yves Bertot and Pierre Castéran. 2004. Interactive theorem proving and program development. Coq’Art: The Calculus of Inductive Constructions. Springer Verlag.

Richard Bornat, Cristiano Calcagno, Peter W. O’Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In ACM Symposium on Principles of Programming Languages (POPL). 259–270.

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. Proving linearizability using forward simulations. In Computer Aided Verification (CAV). 542–563.

Stephen Brookes. 2007. A semantics for concurrent separation logic. Theoretical Computer Science (TCS) 375, 1-3 (2007), 227–270.

Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2014. Parameterised lineairisability. In International Colloquium on Automata, Languages and Programming (ICALP). 98–109.

P. J. Courtois, F. Heymans, and D. L. Parnas. 1971. Concurrent control with “readers” and “writers”. Commun. ACM 14, 10 (1971), 667–668.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In European Conference on Object-Oriented Programming (ECOOP). 207–231.

Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2017. Concurrent data structures linked in time. In European Conference on Object-Oriented Programming (ECOOP). 8:1–8:30.

John Derrick, Gerhard Schellhorn, and Heike Wehrheim. 2011. Mechanically verified proof obligations for linearizability. ACM Transactions on Programming Languages and Systems (TOPLAS) 33, 1 (2011), 4:1–4:43.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent abstract predicates. In European Conference on Object-Oriented Programming (ECOOP). 504–528.

Ivana Filipovic, Peter W. O’Hearn, Noam Rinetzky, and Hongseok Yang. 2010a. Abstraction for concurrent objects. Theoretical Computer Science (TCS) 411, 51–52 (2010), 4379–4398.
Specifying Concurrent Programs in Separation Logic: Morphisms and Simulations

Ivana Filipovic, Peter W. O’Hearn, Noah Torp-Smith, and Hongseok Yang. 2010b. Blaming the client: on data refinement in the presence of pointers. Formal Aspects of Computing 22, 5 (2010), 547–583.

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: a mechanised relational logic for fine-grained concurrency. In IEEE Symposium on Logic in Computer Science (LICS). 442–451.

Alexey Gotsman and Hongseok Yang. 2012. Linearizability with ownership transfer. In International Conference on Concurrency Theory (CONCUR). 256–271.

Daniel Gratzer, Aleš Bizjak, Robbert Krebbers, and Lars Birkedal. 2019. Iron: managing obligations in higher-order concurrent separation logic. In ACM Symposium on Principles of Programming Languages (POPL). to appear.

Ronghui Gu, Jérémi Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep specifications and certified abstraction layers. In ACM Symposium on Principles of Programming Languages (POPL). 595–608.

Ronghui Gu, Zhong Shao, Jeung Kim, Xiongnan (Newman) Wu, Jérémi Koenig, Wilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In ACM Conference on Programming Languages Design and Implementation (PLDI). 646–661.

Ichiro Hasuo, Chris Heunen, Bart Jacobs, and Ana Sokolova. 2009. Coalgebraic Components in a Many-Sorted Microcosm. In Algebra and Coalgebra in Computer Science. 64–80.

Danny Hendler, Itai Ince, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). 355–364.

Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2013. Aspect-oriented linearizability proofs. In International Conference on Concurrency Theory (CONCUR). 242–256.

Maurice Herlihy and Nir Shavit. 2008. The art of multiprocessor programming. M. Kaufmann.

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Computer Systems (TOCS) 12, 3 (1990), 463–492.

C. A. R. Hoare. 1972. Towards a theory of parallel programming. In Operating Systems Techniques. 61–71.

Bart Jacobs. 2016. Introduction to coalgebra: towards mathematics of states and observation. Cambridge University Press.

Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In ACM Symposium on Principles of Programming Languages (POPL). 271–282.

Radha Jagadeesan and James Riely. 2014. Between linearizability and quiescent consistency - quantitative quiescent consistency. In International Colloquium on Automata, Languages and Programming (ICALP). 220–231.

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. ACM Trans. Comput. Syst. 5, 4 (1983).

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: a modular foundation for higher-order concurrent separation logic. Journal of Functional Programming (JFP) (2018). To appear.

Ralf Jung, David Swasey, Filip Siewczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In ACM Symposium on Principles of Programming Languages (POPL). 637–650.

Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. 2017. Proving linearizability using partial orders. In European Symposium on Programming (ESOP). 659–667.

Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective auxiliary state for coarse-grained concurrency. In ACM Symposium on Principles of Programming Languages (POPL). 561–574.

Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In ACM Conference on Programming Languages Design and Implementation (PLDI). 459–470.

Hongjin Liang and Xinyu Feng. 2018. Progress of concurrent objects with partial methods. In ACM Symposium on Principles of Programming Languages (POPL). 20:1–20:31.

Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In ACM Symposium on Principles of Programming Languages (POPL). 455–468.

Hongjin Liang, Xinyu Feng, and Zhong Shao. 2014. Compositional verification of termination-preserving refinement of concurrent programs. In IEEE Symposium on Logic in Computer Science (LICS). 65:1–65:10.

John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. ACM Transactions on Computer Systems (TOCS) 9, 1 (1991), 21–65.

Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2019. PCM and Resource Morphisms for Specifying Concurrent Programs in Separation Logic: supplemental material including the Coq source code. (2019). Available on request.

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating state transition systems for fine-grained concurrent resources. In European Symposium on Programming (ESOP). 290–310.

Peter W. O’Hearn. 2007. Resources, concurrency, and local reasoning. Theoretical Computer Science (TCS) 375, 1-3 (2007).
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 2 and 4. For that, we need first to give some basic definitions for the product of PCMs and the coupling of transitions.

**Lemma A.1.** 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 $M(Spin)$ and $M(Counter)$, that is, $M(\text{SC}) = M(Spin) \times M(\text{CSL})$.

**Definition A.2.** Let $s_i = (a_i^1, a_i^j, a_i^o)$ 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 $[s_1, s_2] \equiv ((a_1^1, a_2^1), (a_1^j, a_2^j), (a_1^o, a_2^o))$, and projection states of $s$ are $s \upharpoonright i \equiv (a_s \upharpoonright i, a_j \upharpoonright i, a_o \upharpoonright i)$, $i = 1, 2$. The usual beta and eta laws for products hold, i.e.: $[s_1, s_2] \upharpoonright 1 = s_1$ and $s = [s\upharpoonright 1, s\upharpoonright 2]$.

**Definition A.3.** Given state spaces $\Sigma_1$ and $\Sigma_2$, the product state space $\Sigma_1 \times \Sigma_2$ is defined by:

$$r_{\upharpoonright \Sigma} \equiv r_{\upharpoonright \Sigma_1} \land r_{\upharpoonright \Sigma_2}$$

$s \in \Sigma_1 \times \Sigma_2$ iff $s \upharpoonright 1 \in \Sigma_1 \land s \upharpoonright 2 \in \Sigma_2 \land$ defined $r_{\upharpoonright \Sigma}$.

**Definition A.4.** Let $t_i$ be a $\Sigma_i$-transition, $i = 1, 2$. The coupling of $t_1$ and $t_2$ is the $\Sigma_1 \times \Sigma_2$-transition defined as $(t_1 \Rightarrow t_2)$ $s' \equiv t_1$ ($s' \upharpoonright 1$) $\land$ $t_2$ ($s' \upharpoonright 2$) ($s' \upharpoonright 2$) 20.

The coupling of transitions is used extensively in our constructions, e.g. in lock_tr $\Rightarrow$ open_tr in Section 2. As we have already explained there, the formalization in Definition A.4 entails that the coupling of two transitions execute their components simultaneously: lock_tr from Spin and open_tr from Counter in the aforementioned example.

Now we can formalize the notion of combined resource, which the next definition makes precise.

**Definition A.5.** Let $V, W$ be resources, $I \subseteq \Delta(V) \times \Delta(W)$ a relation between transitions, such that for every $(t_v, t_w) \in I$ the transition $t_v \Rightarrow t_w$ is footprint-preserving. The combined (alt. tensor) resource $V \otimes I W$ has the state space $\Sigma(V \otimes I W) = \Sigma(V) \times \Sigma(W)$ and transitions:

$$\Delta(V \otimes I W) = \{t_v \Rightarrow t_w \mid (t_v, t_w) \in I\}$$

For example, resource SC is the combination of resources Spin and Counter, with $I$ defined as:

$$I = \{(\text{lock_tr, open_tr}), (\text{unlock_tr, close_tr}), (\text{id_tr, id_tr})\}$$

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

B  SUB-PCMS, COMPATIBILITY RELATIONS, AND PCM MORPHISMS

In Section 4, we implicitly used a construction to restrict the state space of the resource CSL‘ with an invariant $Inv$, to obtain the desired resource CSL. In this section, we formally develop that construction, which hinges on the concept of sub-PCMs. A sub-PCM is to a PCM, what subset is to a set. It is customary in abstract algebra to develop the notion of a sub-object of an algebraic object by first developing the appropriate notion of morphism on the objects; in our case, this will be PCM morphisms. The morphisms will model the injection of a sub-PCM into the PCM (similar to how one can inject a subset into a larger set), and a retraction from the larger PCM into the sub-PCM.

20In this paper we only consider footprint preserving transitions. In Coq code we consider more general notions of transitions which need not be footprint preserving. For that case, we somewhat modify the definition of coupling to require an additional conjunct that $r_{\upharpoonright \Sigma}$ is a defined heap, similar to the definition of product state spaces.
We start with the auxiliary notion of a *compatibility relation*. A compatibility relation serves to delineate the elements of the larger PCM that we want to include into the sub-PCM. Interestingly, compatibility relations are *binary* relations, rather than simple unary predicates over the domain of the larger PCM. This is so, because the sub-PCM construction requires not only selecting the elements out of the larger PCM, but also restricting its join operation, i.e., make it ever more partial than it already is. As the join operation is itself binary, so must be the compatibility relation. One may thus think of the compatibility relation as being an abstract algebraic generalization of the idea of *disjointness* (and indeed, disjointness of histories and heaps are instances of compatibility relations on the PCMs of histories and heaps, respectively).

**Definition B.1.** Given a PCM $A$, relation $R$ on $A$ is a *compatibility relation* if it is:

1. (unitary) $\mathbb{1}_A R \mathbb{1}_A$
2. (commutative) $x R y$ iff $y R x$
3. (compatible) if $x R y$ then $x \perp y$
4. (unital) if $x R y$ then $x R \mathbb{1}_A$
5. (associative) if $x R y$ and $(x \bullet y) R z$ then $x R (y \bullet z)$ and $y R z$

For example, the relation $\perp_{\omega}$ defined in Section 4 in clause (3) on page 16, as


x \perp_{\omega} y = \begin{cases} \text{last\_op\_x = L} & \text{last\_stamp\_y < last\_stamp\_x) \land} \\
\text{last\_op\_y = L} & \text{last\_stamp\_x < last\_stamp\_y) \land x \perp y} 
\end{cases}

is a compatibility relation. The critical part of the proof is that of associativity, which we briefly sketch. Let $x$, $y$, and $z$ be three histories such that $x \perp_{\omega} y$ and $(x \bullet y) \perp_{\omega} z$, and let the timestamps of their last entries be $t_x$, $t_y$, and $t_z$, respectively. The interesting case is when $x$ or $y$ end with a locking entry. Without loss of generality, let last\_op\_x = L. Then, by $x \perp_{\omega} y$, the last entry of $y$ must be $U$, and $t_y < t_x$. Similarly, by $(x \bullet y) \perp_{\omega} z$, the last entry of $z$ must be $U$ and $t_z < t_x$. But then, trivially, $y \perp_{\omega} z$, because both $y$ and $z$ end with an $U$ entry. Similarly, $x \perp_{\omega} (y \bullet z)$, because $y \bullet z$ also ends with an $U$ entry, and its last timestamp is $\max(t_y, t_z) \leq t_y, t_z < t_x$.

Similarly, the clauses (1) and (2) in Section 4 also arise as instances of compatibility relations, and thus, ultimately, the invariant $\text{Inv} = (1) \land (2) \land (3)$ can be defined as a compatibility relation on the PCM $M$ (CSL').

Now we can define PCM morphisms.

**Definition B.2.** A *PCM morphism* $\phi : A \to B$ with a compatibility relation $\perp_{\phi}$ is a partial function from $A$ to $B$ such that:

1. $\phi \mathbb{1}_A = \mathbb{1}_B$
2. if $x \perp_{\phi} y$, then $\phi x, \phi y$ exist, and $\phi x \perp_B \phi y$, and $\phi (x \bullet y) = \phi x \bullet \phi y$

The morphism $\phi$ is *total* if $\perp_{\phi}$ equals $\perp_A$.

In Section 4 we have silently used many PCM morphisms, as the special symbols that we used to name the components such as $\pi$, $\tau$, $\mu$, $\sigma$, etc., are all PCM morphisms. For example, $\tau$ and $\pi$ in Figure 5 are projections out of the PCM $M$ (Spin) into the PCM of histories and permissions, respectively (and projections out of tuples are morphisms). Then, $\tau_s s$ and $\tau_o s$ are abbreviations for $\tau(a_s s)$ and $\tau(a_o s)$ respectively, and similarly for $\pi$.

Now we define the notion of *sub-PCM* of another PCM.

**Definition B.3.** PCM $A$ is a *sub-PCM* of a PCM $B$ if there exists a total PCM morphism $\iota : A \to B$ (injection) and a morphism $\rho : B \to A$ (retraction), such that:

1. $\rho (\iota a) = a$
2. if $b \perp_{\rho} \mathbb{1}_B$ then $\iota (\rho b) = b$
(3) if \((\rho x) \perp_A (\rho y)\) then \(x \perp_{\rho} y\)

Property (1) says that \(\iota\) is injective, i.e., if we coerce \(a \in A\) into \(a\), we can recover \(a\) back by \(\rho\), since no other element of \(A\) maps by \(\iota\) into \(a\). The dual property (2) allows the same for a subset of \(B\)'s elements, that are related by \(\perp_{\rho}\) to \(1_B\). Hence, intuitively, \(A\) is in 1-1 correspondence with that subset of \(B\). Property (3) extends the correspondence to compatibility relations, i.e., \(\perp_A\), when considered on images under \(\rho\), implies (and hence, by properties of morphisms, equals) \(\perp_{\rho}\).

Definition B.3 says what it means for \(A\) to be a sub-PCM of \(B\). The following lemma shows how to construct a sub-PCM of \(B\) given a compatibility relation \(R\) on \(B\). We used this construction in Section 4 to obtain the PCM for the resource CSL out of the PCM \(M\) (CSL').

**Lemma B.4.** Given PCM \(A\) and compatibility relation \(R\) on \(A\), the set \(A/R = \{x \in A \mid x R 1_A\}\) forms a PCM under unit \(1_A\), and join operation defined as \(x \bullet_{A/R} y = x \bullet_A y\) if \(x R y\) and undefined otherwise. The PCM \(A/R\) is a sub-PCM of \(A\) under the injection \(\iota\) and retraction \(\rho\) defined as \(\forall x \in A/R, \iota(x) = x\) and \(\forall a \in A, \rho(a) = a\) if \(a R 1_A\), and \(\rho(a)\) undefined otherwise. Moreover, \(R = \perp_{\rho} = \perp_{A/R}\).

Using the above notation, the PCM of the resource CSL can be defined formally as \(M\) (CSL')/\(\text{Inv}\).

We now have all the ingredients to formalizing the construction for restricting a resource that we set out to define.

**Definition B.5.** Let \(R\) be an invariant compatibility relation on \(M(V)\). The sub-resource \(V/R\) is defined with the same type, transitions and erasure as \(V\), but with the PCM and the state space defined as

1. \(M(V/R) = M(V)/R\)
2. \(s \in \Sigma(V/R)\) iff \(s \in \Sigma(V) \land (a_s, s) \in R(a_o, s)\)

There is a generic resource morphism \(\iota : V \rightarrow V/R\) that is inclusion on states and identity on transitions.

For example, the resource CSL from Section 4 is the sub-resource of CSL' taken under the invariant \(\text{Inv}\), and the resource morphism \(\iota : \text{CSL'} \rightarrow \text{CSL}\) from Section 4 (page 17), is the generic morphism defined above.

We finish this section by providing some additional evidence that compatibility relations and PCM morphisms compose, and have pleasant mathematical properties. For example, the operations of morphism composition and join come with the compatibility relations as follows.

\[
(f \circ g)(x) \cong f(gx) \quad x \perp_{fg} y \cong x \perp g y \land gx \perp f g y \\
(f \bullet g)(x) \cong f x \bullet g x \quad x \perp_{fg} y \cong x \perp f y \land x \perp g y \land f(x \bullet y) \perp g(x \bullet y)
\]

Or, given PCM morphisms \(f\) and \(g\), we can define compatibility relation that implements the PCM versions of the algebraic notions of kernel (preimages of unit) and equalizer (values on which two morphisms agree), as follows.

\[
x (\ker f) y \cong x \perp f y \land f x = f y = 1 \\
x (\text{eql} f g) y \cong x \perp f y \land x \perp g y \land f x = g x \land f y = g y
\]

Importantly, the above are all compatibility relations, as we have proved in the Coq files. Similarly, we can restrict a morphism to a compatibility relation \(R\), to define another PCM morphism.

\[
(f/R)(x) \cong \begin{cases} 
  f x, & \text{if } x R 1 \\
  \text{undefined}, & \text{otherwise}
\end{cases}
\text{ with } x \perp_{f/R} y \cong x \perp f y \land x R y
\]

The import of the above abstract constructions is in the reduction of proof burden. For example, a morphism equalizer is a compatibility relation by construction, so the user need not bother proving compatibility for equalizers. The constructions also combine to concisely state invariants.
and assertions. For example, the compatibility relation that gives rise to \( Inv \), and is thus used to construct the sub-resource CSL, may be defined as the equalizer \( eql (\pi \bullet \alpha) (\omega' \circ \tau) \), where \( \omega' : \text{hist} \to O \) is the morphism defined on a history \( h \) as \( \omega' h = \text{if } \omega h \) then own else \( \overline{\text{own}} \).

### C 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 3.

**Definition C.1.** An **indexed family of morphisms** \( f : V \xrightarrow{\Sigma} W \) (or just *family*), consists of partial functions \( f_x : X \to \Sigma(W) \to \Sigma(V) \) (note the contravariance), and \( f_{\Delta} : X \to \Delta(V) \to \Delta(W) \) on transitions, such that:

1. (locality of \( f_x \)) there exits a function \( f_A : M(W) \to M(V) \) such that if \( f_x x (s_w \triangleright p) = s_u \), then \( s_u = s'_u \triangleright f_A(p) \), and \( f_x x (s_w \prec p) = s'_u \prec f_A(p) \).
2. (locality of \( f_{\Delta} \)) if \( f_x x (s_w \triangleright p) t = t' \), then \( f_{\Delta} x (s_w \prec p) t = t' \).
3. (other-fixity) if \( a_o(s_w) = a_o(s'_w) \) and \( f_x x s_w, f_x x' s'_w \) exist, then \( a_o(f_x x s_w) = a_o(f_x x' s'_w) \).
4. (injectivity on indices) if \( f_x x_1 s_{w_1} \) and \( f_x x_2 s_{w_2} \) exist and \( f_x x_1 s_{w_1} = f_x x_2 s_{w_2} \), then \( x_1 = x_2 \).

In most of the properties of Definition C.1, the index \( x \) is simply propagated unchanged. The only property where \( x \) is significant is the new property (4), which requires that \( f_x x s_w \) uniquely determines the index \( x \). In the Stack example in Section 5, it is easy to see that the definition of \( f_x \) satisfies this property, because equal states have equal histories.

**Definition C.2.** Given a morphism family \( f : V \xrightarrow{\Sigma} W \), an *\( f \)-simulation* is a predicate \( I \) on \( X \) and \( W \)-states such that:

1. if \( I x s_w \), and \( f_x x s_w = s_u \), and \( t s_u s'_u \), then there exist \( x', t' = f_{\Delta} x s_w t \), and \( s'_w \) such that \( I x' s'_w \) and \( f_x x' s'_w = s'_u \), and \( t' s_w s'_w \).
2. if \( I x s_w \), and \( s_u \rightarrow V s'_u \), then \( I x s'_w \), and \( s'_w = f_x x s'_{w} \) exists, and \( s_u \rightarrow W s'_w \).

Compared to Definition 3.9, property (1) allows that \( x \) changes into \( x' \) by a transition. In the Stack example in Section 5, if we lift \( e \) by using the index \( x = \emptyset \) (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.

### D Denotational Semantics

Our semantic model largely relies on the denotational semantic of *action trees* [Ley-Wild and Nanevski 2013]. A tree implements a finite partial approximation of one execution of a program. Thus, a program of type \( ST V A \) 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 [Brookes 2007] 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 parametrized 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 D.3) 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 3, and from there, in defining the type of Hoare triples \( \{P\} A \{Q\}@V \). Both the \( STV A \) 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 FCSSL follows from showing that the lemmas about the vrf predicate transformer listed in Section 3, are satisfied by the denotations of the commands.

We choose the Calculus of Inductive Constructions (CiC) [Bertot and Castéran 2004; The Coq Development Team 2018] 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 [The Coq Development Team 2018], 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.

**D.1 Action trees and program denotations**

**Definition D.1 (Action trees).** The type tree \( VA \) of (A-returning) action trees is defined by the following iterated inductive definition.

\[
\text{tree } VA \equiv \begin{align*}
\text{Unfinished} \\
| \text{Ret } (v : A) \\
| \text{Act } (a : \text{action } VA) \\
| \text{Seq } (T : \text{tree } VB) (K : B \rightarrow \text{tree } VA) \\
| \text{Par } (T_1 : \text{tree } VB_1) (T_2 : \text{tree } VB_2) (K : B_1 \times B_2 \rightarrow \text{tree } VA) \\
| \text{Morph } (f : W \rightarrow V) (x : X) (T : \text{tree } WA)
\end{align*}
\]

Most of the constructors in Definition D.1 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 \( a : \text{action } VA \), as defined in Section 3. 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 \( B_1 \) 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 \( f : W \rightarrow V \), and tree \( T : \text{tree } WA \) 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 \( STVA \) of sets of trees, containing Unfinished.

\[
\text{ST } VA \equiv \{e : \text{set } (\text{tree } VA) | \text{Unfinished} \in e\}
\]
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 $\{\text{Unfinished}\} \parallel e'$ is denoted by the set:

$$\{\text{Unfinished}\} \cup \{\text{Par Unfinished} \ T \ \text{Ret} \mid T \in e'\}$$

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

---

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

\[
\Delta \vdash x, s, \text{Seq} \ (\text{Ret} \ v) \ K \xrightarrow{\text{SeqRet}} x, s, K \ v
\]

\[
\Delta \vdash x, s, \text{Par} \ (\text{Ret} \ v_1) \ (\text{Ret} \ v_2) \ K \xrightarrow{\text{ParRet}} x, s, K (v_1, v_2)
\]

\[
\Delta \vdash x, s, T_1 \ T_2 \xrightarrow{\pi} x', s', T_2
\]

\[
\Delta \vdash x, s, \text{Par} \ T_1 \ T_2 \ K \xrightarrow{\text{ParL} \ \pi} x', s', \text{Par} T_1 \ T_2
\]

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

\[
\Delta \vdash x, s, \text{Ret} \ v \xrightarrow{\pi} x', s', T'
\]

\[
\Delta \vdash x, s, \text{Act} \ a \xrightarrow{\text{ChoiceAct} \ \pi} x', s', \text{Ret} v
\]

\[
\Delta \vdash x, s, \text{Morph} \ f \ y \ (\text{Ret} \ v) \xrightarrow{\text{MorphRet}} x, s, \text{Ret} v
\]

\[
\Delta, f \vdash (x, y), s, T \xrightarrow{\pi} (x', y'), s', T'
\]

\[
\Delta \vdash x, s, \text{Morph} \ f \ T \xrightarrow{\text{MorphStep} \ \pi} x', s', \text{Morph} f \ y' \ T'
\]

Fig. 6. Judgment $\Delta \vdash x, s, T \xrightarrow{\pi} x', s', T'$, for operational semantics on trees, which reduces a tree with respect to the path $\pi$. 

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

\[
\text{ret } (r : A) \equiv \{\text{Unfinished}, \text{Ret} r\}
\]

\[
x \leftarrow e_1; e_2 \equiv \{\text{Unfinished}\} \cup \{\text{Seq} T_1 \ K \mid T_1 \in e_1 \land \forall x. K \ x \in e_2\}
\]

\[
e_1 \parallel e_2 \equiv \{\text{Unfinished}\} \cup \{\text{Par} T_1 \ T_2 \ \text{Ret} \mid T_1 \in e_1 \land T_2 \in e_2\}
\]

\[
\langle a \rangle \equiv \{\text{Unfinished}, \text{Act} a\}
\]

\[
\text{morph } f x \ e \equiv \{\text{Unfinished}\} \cup \{\text{Morph} f x \ T \mid T \in e\}
\]
D.2 Operational semantics of action trees

The judgment for small-step operational semantics of action trees has the form \( \Delta \vdash x, s_w, T_v \xrightarrow{\pi} x', s'_w, T'_v \) (Figure 6). 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

\[
f_0 : V_1 \xrightarrow{X_0} W, f_1 : V_2 \xrightarrow{X_1} V_1, \ldots, f_n : V_n \xrightarrow{X_n} V_n
\]

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 \( x \) and \( x' \) 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

\[
morph f_0 x_0 \cdot (\cdots (\text{morph } f_n x_n e) \cdots),
\]

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 \( T_v, T'_v : \text{tree } V A \), 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.

\[
\text{path} \equiv \text{ChoiceAct} | \text{SeqRet} | \text{SeqStep} (\pi : \text{path}) | \text{ParR} (\pi : \text{path}) | \text{ParL} (\pi : \text{path}) | \text{MorphStep} (\pi : \text{path})
\]

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 \( \text{ComSquares} \Delta \times X^* s_v, s'_v, t_v, s_w, s'_w, t_w \) 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 \( \Delta = f_0; f_1; \ldots; f_n \)).

![Diagram](image-url)
The relation captures that $t_u$ 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 $x$ or type $(X_0, \cdots, X_n)$, until we reach the state $s_v$. Then transition $t_u$ is executed in $s_v$ to obtain $s'_v$, and the transition components of the morphisms are run backwards to obtain $t_w$, and the associated end state $s'_w$. Formally:

$$\text{ComSquares} \cdot () () s_v s'_v t_u s_w s'_w t_w =$$

$$s_v = s_w \land s'_v = s'_w \land t_u = t_w$$

$$\text{ComSquares}(f_0 : V_1 \rightarrow W), (x_0, x) (x'_0, x')$$

$$s_v s'_v t_u s_w s'_w t_w =$$

$$\exists s_1 s'_1 t_1. \text{ComSquares} (\bar{x} \bar{x'}) s_v s'_v t_u s_1 s'_1 t_1 \land$$

$$f_0 x_0 s_w = s_1 \land f_0 x'_0 s'_w = s'_1$$

We will often abbreviate ComSquares, and write:

$$\text{unwind}_\text{tr} \Delta \bar{x} \bar{x'} t_v s_w s'_w =$$

$$\exists s_v s'_v t_w. \text{ComSquares} \Delta \bar{x} \bar{x'} s_v s'_v t_u s_w s'_w t_w$$

$$\text{unwind}_\text{act} \Delta \bar{x} \bar{x'} a s_w s'_w v =$$

$$\exists s_v s'_v t_u t_w. a s_v = (t_u, v) \land$$

$$\text{ComSquares} \Delta \bar{x} \bar{x'} s_v s'_v t_u s_w s'_w t_w$$

Given a context $\Delta$, input indices $\bar{x}$, input state $s_w$ and input transition $t_u$, `unwind_tr` "unwinds" the transition, in the sense that it computes the output index $\bar{x}'$, output state $s'_w$, obtained by the morphing iteratively by the morphisms in $\Delta$. The abbreviation unwind_act does the same, but for input action $a : \text{actionVA}$. In the process, unwind_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 6, and simply write $s, t \rightarrow s', T'$.

The operational semantics on trees in Figure 6 may not make a step on a tree for two different reasons. The first, benign, reason is that the 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 satisfies. For example, in the ChoiceAct rule, there may not exist a $v$ such that unwind $\Delta \bar{x} \bar{x'} a s s' v$. To differentiate between these two different reasons, we first define the notion of well-formed, or good path, for a given tree.

**Definition D.2 (Good paths).** Let $T : \text{tree VA}$ and $\pi$ be a path. Then the predicate good $T \pi$ recognizes good paths for a tree $T$ as follows:

- good $(\text{Act } a) \text{ ChoiceAct } \equiv \text{true}$
- good $(\text{Seq } (\text{Ret } v) _) \text{ SeqRet } \equiv \text{true}$
- good $(\text{Seq } T _) \text{ SeqRet } \equiv \text{true}$
- good $(\text{Par } (\text{Ret } _) (\text{Ret } _) \text{ParRet } \equiv \text{true}$
- good $(\text{Par } T_1 T_2 _) \text{ ParL } \equiv \text{good } T_1 \pi$
- good $(\text{Par } T_1 T_2 _) \text{ ParR } \equiv \text{good } T_2 \pi$
- good $(\text{Morph } f x (\text{Ret } _)) \text{ MorphRet } \equiv \text{true}$
- good $(\text{Morph } f x T) \text{ MorphStep } \equiv \text{good } T \pi$
- good $T \equiv \text{false otherwise}$
Definition D.3 (Safe configurations). We say that a state \( s \) is \textit{safe} for the tree \( T \) and path \( \pi \), written \( s \in \text{safe} \ t \ \pi \) if:
\[
\text{good } T \pi \rightarrow \exists \pi' \ T'. \ s, T \overset{\pi}{\rightarrow} s', T'
\]

Notice that in the above definition, the trees Unfinished and Ret \( \nu \) 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.

Lemma D.4 (Coverage of stepping by transitions). Let \( \Delta : V \rightarrow W \), and \( \Delta \vdash \bar{x}, s, T \overset{\pi}{\rightarrow} \bar{x}', s', T' \). Then either the step corresponds to an idle transition, that is, \( (\bar{x}, s, t) = (\bar{x}', s') \), or there exists a transition \( a \in \Delta (V) \), such that unwind_tr \( \Delta \bar{x} \bar{x}' \ a \ s \bar{s}' \ \nu \).

Lemma D.5 (Other-fixity of stepping). Let \( \Delta : V \rightarrow W \) and \( \Delta \vdash \bar{x}, s, T \overset{\pi}{\rightarrow} \bar{x}', s', T' \). Then \( a_0 (s) = a_0 (s') \).

Lemma D.6 (Stepping preserves state spaces). Let \( \Delta : V \rightarrow W \) and \( \Delta \vdash \bar{x}, s, T \overset{\pi}{\rightarrow} \bar{x}', s', T' \). If \( s \in \Sigma (W) \) then \( s' \in \Sigma (W) \).

Lemma D.7 (Stability of stepping). Let \( \Delta : V \rightarrow W \) and \( \Delta \vdash \bar{x}, s, T \overset{\pi}{\rightarrow} \bar{x}', s', T' \). Then \( s \overset{T}{\rightarrow} s' \).

Lemma D.8 (Determinism of stepping). Let \( \Delta : V \rightarrow W \) and \( \Delta \vdash \bar{x}, s, T \overset{\pi}{\rightarrow} \bar{x}', s', T', \) and \( \Delta \vdash \bar{x}, s, T \overset{\pi}{\rightarrow} \bar{x}'', s'', T'' \). Then \( \bar{x}' = \bar{x}'' \), \( s' = s'' \), and \( T' = T'' \).

Lemma D.9 (Locality of stepping). Let \( \Delta : V \rightarrow W \) and \( \Delta \vdash \bar{x}, (s \triangleright p), T \overset{\pi}{\rightarrow} \bar{x}', s', T' \). Then there exists \( s'' \) such that \( s' = s'' \triangleright p \), and \( \Delta \vdash \bar{x}, (s \triangleleft p), T \overset{\pi}{\rightarrow} \bar{x}', (s'' \triangleleft p), T' \).

Lemma D.10 (Safety-monotonicity of stepping). If \( s \triangleright p \in \text{safe } T \), then \( s \triangleleft p \in \text{safe } T \).

Lemma D.11 (Frameability of stepping). Let \( s \triangleleft p \in \text{safe } T \), and \( s \triangleright p, T \overset{\pi}{\rightarrow} s', T' \). Then there exists \( s'' \) such that \( s' = s'' \triangleright p \) and \( s \triangleright p, T \overset{\pi}{\rightarrow} s'', T' \).

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 \( f : V \rightarrow W \).

Lemma D.12 (Stepping under morphism). Let \( f : V \rightarrow WX \) and \( f \bar{x} s_w = s_u \). Then the following hold:

1. If \( s_w, T \overset{\pi}{\rightarrow} s'_w, T' \), then \( \exists \bar{x}' s'_w \). \( f \bar{x}' s'_w = s'_u \) and \( f \vdash \bar{x}, s_w, T \overset{\pi}{\rightarrow} \bar{x}' s'_w, T' \).
2. If \( f \vdash \bar{x}, s_w, T \overset{\pi}{\rightarrow} \bar{x}' s'_w, T' \), then \( \exists \bar{x}' s'_w \). \( f \bar{x}' s'_w = s'_u \) and \( s_u, T \overset{\pi}{\rightarrow} s'_u, T' \).

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 \( f \bar{x}' \)’s state component is a function in the contravariant direction. Thus, for each \( s_w \) there are unique \( x \) and \( s_u \), such that \( f \bar{x} s_w = s_u \).
D.3 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 D.13 (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:

\[
\text{always}^\zeta T X s = \begin{cases} 
\forall s_2, (s \xrightarrow[V]{} s_2) \rightarrow \text{safe } T \pi s_2 \land X s_2 T \land \\
\forall s_3 T'. (s_2, T \xrightarrow[\pi]{} s_3, T') \rightarrow \text{always}^\zeta' T_2 X s_3 
\end{cases}
\]

\[
\text{always } T X s = \forall \zeta. \text{always}^\zeta T X s
\]

\[
\text{after} T Q = \text{always } (\lambda s' T', \forall v. T' = \text{Ret } v \rightarrow Q v s')
\]

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.

\[
vrf e Q s = \forall T \in e. \text{after } T Q s
\]

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 3 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 D.14 (Soundness).

- All the properties of vrf predicate transformer from Section 3 are valid.
- The sets 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.
- All program constructors are monotone.