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 set[1]:

$$\mathsf{lock} \;\widehat{=}\; \mathsf{do}\ x \leftarrow \mathsf{CAS}(r, \mathsf{false}, \mathsf{true})\ \mathsf{while}\ \neg x$$

The program isn't atomic because it performs potentially many compare-and-set (CAS) commands in a loop. However, it can be viewed as *abstractly atomic* by clients, for whom it's enough to observe the effect of lock acquisition (i.e., CAS succeeds) and to ignore all previous, unsuccessful, loop iterations. Verifying such clients typically requires associating lock with some ghost code α that operates over the client's own state, and executes *simultaneously* with the lock's effect, essentially to inform the client that locking took place. The problem of abstract atomicity is how to formally represent and reason about this simultaneity. The state of the art [Jacobs and Piessens 2011; Jung et al. 2015; Svendsen and Birkedal 2014; Svendsen et al. 2013] is to parametrize lock by α, *a priori*, thus modifying it into:

$$\begin{aligned}\mathsf{lock}\ \alpha \;\widehat{=}\; & \mathsf{do}\ \langle x \leftarrow \mathsf{CAS}(r, \mathsf{false}, \mathsf{true});\\ & \qquad \boxed{\mathsf{if}\ x\ \mathsf{then}\ \alpha}\ \rangle\ \mathsf{while}\ \neg x\end{aligned}$$

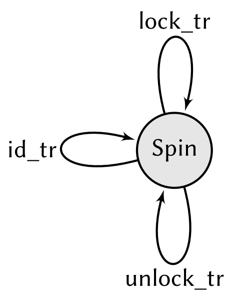
where ghost code is given in gray background, and brackets indicate atomic execution during which other threads can't run[2]. This makes the verification compositional, as one can verify lock α out of the spec of α. We discuss such approaches further in Section 6.

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 id_tr, corresponding to unsuccessful CAS's, followed by a locking transition lock_tr. A morphism can re-interpret lock_tr, e.g., to execute α in addition to locking, thus achieving the desired simultaneity.

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

Referring to the commuting diagram in Figure 1(b), the morphed program is in W, so we describe it starting with a W-state s_w in the lower-right corner of the diagram (the predicate I that is applied to s_w in the diagram will be explained promptly). To compute the next state of the morphed program, we first take $s_v = f_\Sigma\ s_w$ which is a state in V (explaining contravariance of f_Σ). 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.

[1]We assume that 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.
[2]As CAS is already atomic, and α is ghost code, the granularity of the program, and its behavior on real (i.e., non-ghost) state, remain unchanged.



(a)

$$\begin{tikzcd}
\vdots & & \vdots \\
s'_v & & I\, s'_w \arrow[ll, "f_\Sigma"'] \\
s_v \arrow[u, "t_v"] & \arrow[ul, phantom, "\llcorner"] & I\, s_w \arrow[ll, "f_\Sigma"] \arrow[u, "t_w{=}f_\Delta\, s_w\, t_v"']
\end{tikzcd}$$

(b)

Fig. 1. (a) Resource for spin locks, and (b) Reinterpreting transitions by a morphism $f : V \longrightarrow W$.

The process determines a program in W that we denote morph $f\ e$. Morph is a *program constructor*, and a form of *function application* of f to e. As Section 2 illustrates, in the lock example, the resource V is Spin, and W is a combination of Spin and the client resource. The f_Σ component of the morphism projects a Spin state out of the combined state of Spin and the client, while f_Δ maps the lock_tr transition of Spin to a transition that combines lock_tr and α.

1.2 Simulations

To reason about 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 morph $f\ e$ compositionally out of e's *specification*, with the following *single* inference rule, which is our second contribution:

$$\frac{e : \{P\}\{Q\}@V}{\text{morph } f\ e : \{\lambda s_w.\, f\hat{\ }P\, s_w \wedge I\, s_w\}\{\lambda s_w.\, f\hat{\ }Q\, s_w \wedge I\, s_w\}@W}$$

Here $f\hat{\ }P\, s_w$ is defined as $\exists s_v.\, s_v = f_\Sigma\, s_w \wedge P\, s_v$. 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 [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 $f\hat{\ }Q\, s''_w$) 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 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_Σ.

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 $\tau_s\ s$, 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 τ_s and τ_o as *self* and *other* histories, respectively [Nanevski et al. 2014; Sergey et al. 2015], and denote by $\hat{\tau}\ s$ the *union history* $\tau_s\ s \bullet \tau_o\ s$.

$$\begin{aligned}
\mathsf{lock} : [h,k].\ & \{\lambda s.\ \tau_s\ s = h \wedge k \leq \mathsf{last\_stamp}\ (\hat{\tau}\ s)\} \\
& \{\lambda s.\ \exists t.\ \tau_s\ s = h \bullet t \mapsto \mathsf{L} \wedge k < t\}@\mathsf{Spin} \\
\mathsf{unlock} : [h,k].\ & \{\lambda s.\ \tau_s\ s = h \wedge k \leq \mathsf{last\_stamp}\ (\hat{\tau}\ s)\} \\
& \{\lambda s.\ \exists t.\ \tau_s\ s = h \bullet t \mapsto \mathsf{U} \wedge k < t\ \vee \\
& \qquad \tau_s\ s = h \wedge \hat{\tau}\ s\ t = \mathsf{U} \wedge k \leq t\}@\mathsf{Spin}
\end{aligned}$$

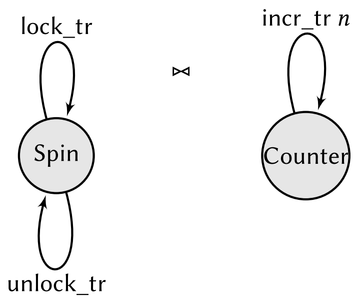
A history is a timestamped log of the locking and unlocking operations. Mathematically, it's a *finite map* from timestamps (strictly positive nats) to the set $\{\mathsf{L}, \mathsf{U}\}$. For example, if the history $\tau_s\ s$ equals $2 \mapsto \mathsf{U} \bullet 7 \mapsto \mathsf{L} \bullet 9 \mapsto \mathsf{L}$, 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 $2 \mapsto \mathsf{U}$ 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[6] in the precondition, which is increased in the postcondition to log a locking event at time t. The conjunct $k < t$ 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-Crummey 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.



State space: $s \in \Sigma\,(\mathsf{Spin})$ iff
$(\tau_s\, s \perp \tau_o\, s) \wedge \mathsf{alternate}\,(\hat{\tau}\, s) \wedge r \neq \mathsf{null}$
Erasure: $\ulcorner s \urcorner \mathrel{\widehat{=}} r \Mapsto \omega\,(\hat{\tau}\, s)$
Transitions $\Delta(\mathsf{Spin})$:
$\mathsf{lock\_tr}\; s\; s' \mathrel{\widehat{=}} \neg\omega\,(\hat{\tau}\, s) \wedge$
$\tau_s\, s' = \tau_s\, s \bullet \mathsf{fresh}\,(\hat{\tau}\, s) \Mapsto \mathsf{L}$
$\mathsf{unlock\_tr}\; s\; s' \mathrel{\widehat{=}} \omega\,(\hat{\tau}\, s) \wedge$
$\tau_s\, s' = \tau_s\, s \bullet \mathsf{fresh}\,(\hat{\tau}\, s) \Mapsto \mathsf{U}$

State space: $s \in \Sigma\,(\mathsf{Counter})$ iff $\top$
Erasure: $\ulcorner s \urcorner \mathrel{\widehat{=}} \emptyset$
Transitions $\Delta\,(\mathsf{Counter})$:
$\mathsf{incr\_tr}\; n\; s\; s' \mathrel{\widehat{=}} \kappa_s\, s' = \kappa_s\, s + n$

Abbreviations: (in the abbreviations below, τ is a bound variable ranging over histories)

$$\mathsf{dom}\,\tau \mathrel{\widehat{=}} \{t \mid \tau\,(t)\ \mathsf{defined}\} \qquad \mathsf{last\_op}\,\tau \mathrel{\widehat{=}} \begin{cases} \tau\,(\mathsf{last\_stamp}\,\tau), & \text{if } \mathsf{last\_stamp}\,\tau \neq 0 \\ \mathsf{U}, & \text{otherwise} \end{cases}$$

$$\mathsf{last\_stamp}\,\tau \mathrel{\widehat{=}} \max\,(\{0\} \cup \mathsf{dom}\,\tau) \qquad \omega\,\tau \mathrel{\widehat{=}} (\mathsf{last\_op}\,\tau = \mathsf{L})$$

$$\mathsf{fresh}\,\tau \mathrel{\widehat{=}} 1 + \mathsf{last\_stamp}\,\tau$$

$$\begin{aligned} \mathsf{alternate}\,\tau \mathrel{\widehat{=}}\ & (\forall t > 0.\, \tau\,(t+1) = \mathsf{L} \longrightarrow \mathsf{even}\,t \wedge \tau\,t = \mathsf{U}) \wedge \\ & (\forall t > 0.\, \tau\,(t+1) = \mathsf{U} \longrightarrow \mathsf{odd}\,t \wedge \tau\,t = \mathsf{L}) \end{aligned}$$

Fig. 2. Spin and Counter resources (with id_tr elided).

postcondition claims that t is fresh, because it's larger than any k generated prior to the call (as $k \leq \mathsf{last\_stamp}\,(\hat{\tau}\, s)$ 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 $\hat{\tau}\, s$ occurred. Notice that the spec is *stable*, i.e., invariant under interference. Intuitively, other threads can't modify the $\tau_s\, s$ field, as it's private to "us". They can log new events into $\tau_o\, s$, which features in the comparison $k \leq \mathsf{last\_stamp}\,(\hat{\tau}\, s)$, 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 ($\hat{\tau}\; s\; t = \mathsf{U}$). 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 $\hat{\tau}\, s$ alternate between L and U, and that r isn't a null pointer. The *erasure* $\ulcorner s \urcorner$ 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:

```
lock = do ⟨$x \leftarrow \mathsf{CAS}(r, \mathsf{false}, \mathsf{true})$;
             if $x$ then $\tau_s\, s := \tau_s\, s \bullet \mathsf{fresh}\,(\hat{\tau}\, s) \Mapsto \mathsf{L}$ ⟩;
          while $\neg x$

unlock = ⟨ $x \leftarrow !r$; $r := \mathsf{false}$;
            if $x$ then $\tau_s\, s := \tau_s\, s \bullet \mathsf{fresh}\,(\hat{\tau}\, s) \Mapsto \mathsf{U}$ ⟩
```

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 $\langle$lock; $\kappa_s\, s := \kappa_s\, s + n$ $\rangle$ but this isn't quite right as the atomic brackets would prevent other programs from running during the iterations of lock's loop. To model that addition occurs simultaneously with the successful CAS of the last iteration in lock, we use morphisms as follows.

First, we "tensor" the resources Spin and Counter, as graphically indicated[8] in Figure 2; that is, we create a new resource SC whose state is a pair of Spin and Counter states, and transitions are lock_tr $\bowtie$ incr_tr n and unlock_tr $\bowtie$ 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:

$$(t_1 \bowtie t_2)\; s\; s' \;\widehat{=}\; t_1\,(s\backslash 1)\,(s'\backslash 1) \wedge t_2\,(s\backslash 2)\,(s'\backslash 2)$$

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

$$\begin{aligned}
(f\ n)_\Sigma\ s &= s\backslash 1\\
(f\ n)_\Delta\ s\ \mathsf{lock\_tr} &= \mathsf{lock\_tr} \bowtie \mathsf{incr\_tr}\ n\\
(f\ n)_\Delta\ s\ \mathsf{unlock\_tr} &= \mathsf{unlock\_tr} \bowtie \mathsf{id\_tr}
\end{aligned}$$

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

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

[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.

$$\begin{tikzcd}
s'\backslash 1 & I_1\, s' \arrow[l, "(f\ 1)_\Sigma"'] \arrow[dl, phantom, "\llcorner", very near start] \\
s\backslash 1 \arrow[u, "\mathsf{lock\_tr}"] & I_1\, s \arrow[l, "(f\ 1)_\Sigma"] \arrow[u, "(f\ 1)_\Delta\ s\ \mathsf{lock\_tr}"']
\end{tikzcd}$$

Fig. 3. Diagram 1(b) specified for the case of $(f\ 1)$, where $(f\ 1)_\Delta\ s\ \mathsf{lock\_tr} = \mathsf{lock\_tr} \bowtie \mathsf{incr\_tr}\ 1$ and $\mathsf{lock\_tr}\ s\ s' \;\widehat{=}\; \neg\omega\,(\hat{\tau}\ s) \wedge \tau_s\,(s') = \tau_s\ s \bullet \mathsf{fresh}\,(\hat{\tau}\ s) \Mapsto \mathsf{L}$.

to obtain a program in the SC resource, which executes $\mathsf{lock\_tr} \bowtie \mathsf{incr\_tr}\ n$ 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

```
morph (f 1) lock;
morph (f 42) unlock;
morph (f 2) lock
```

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

1. $\{\kappa_s\,(s\backslash 2) = n\}$
2. $\{\kappa_s\,(s\backslash 2) = n \wedge \tau_s\,(s\backslash 1) = h\}$
3. morph $(f\ 1)$ lock;
   // $I_1\ s \;\widehat{=}\; \kappa_s\,(s\backslash 2) = n + \sharp_{\mathsf{L}}\,(\tau_s\,(s\backslash 1)) - \sharp_{\mathsf{L}}\,h$
4. $\{\kappa_s\,(s\backslash 2) = n + \sharp_{\mathsf{L}}\,(\tau_s\,(s\backslash 1)) - \sharp_{\mathsf{L}}\,h \wedge \tau_s\,(s\backslash 1) = h \bullet t \Mapsto \mathsf{L}\}$
5. $\{\kappa_s\,(s\backslash 2) = n + 1 \wedge \tau_s\,(s\backslash 1) = h'\}$
6. morph $(f\ 42)$ unlock; // $I_{42}\ s \;\widehat{=}\; \kappa_s\,(s\backslash 2) = n + 1$
7. $\{\kappa_s\,(s\backslash 2) = n + 1 \wedge (\tau_s\,(s\backslash 1) = h' \vee \tau_s\,(s\backslash 1) = h' \bullet t' \Mapsto \mathsf{U})\}$
8. $\{\kappa_s\,(s\backslash 2) = n + 1 \wedge \tau_s\,(s\backslash 1) = h''\}$
9. morph $(f\ 2)$ lock
   // $I_2\ s \;\widehat{=}\; \kappa_s\,(s\backslash 2) = n + 1 + 2(\sharp_{\mathsf{L}}\,(\tau_s\,(s\backslash 1)) - \sharp_{\mathsf{L}}\,h'')$
10. $\{\kappa_s\,(s\backslash 2) = n + 1 + 2(\sharp_{\mathsf{L}}\,(\tau_s\,(s\backslash 1)) - \sharp_{\mathsf{L}}\,h'') \wedge$
    $\tau_s\,(s\backslash 1) = h'' \bullet t'' \Mapsto \mathsf{L}\}$
11. $\{\kappa_s\,(s\backslash 2) = n + 3\}$

We start with the precondition $\kappa_s\,(s\backslash 2) = n$, 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 $f\ 1$, the simulation I_1 states that the counter κ_s increments by the number of *fresh* L-entries in the history ($\sharp_{\mathsf{L}}\ h$ is the number of L-entries in h). For the morphism $f\ 2$, I_2 says that κ_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 $t_v = \mathsf{lock\_tr}$, because lock_tr adds an L entry to τ_s, and $(f\ 1)_\Delta\ s\ \mathsf{lock\_tr} = \mathsf{lock\_tr} \bowtie \mathsf{incr\_tr}\ 1$ increments κ_s by 1.

But, I_1 is also preserved by unlock_tr, as unlock_tr doesn't add L entries, and $(f\ 1)_\Delta\ s$ unlock_tr = unlock_tr $\bowtie$ id_tr doesn't change κ_s.

It's important that being a simulation is a property independent of the morphed program, as then the proofs can be derived *compositionally*, i.e. from the program *spec*, not code. Specifically above, each of I_1, I_2 and I_{42} enables computing the end-value of κ_s from the end-value of τ_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., τ of Spin and κ 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 $s = (a_s, a_j, a_o)$ is an (M, T)-state *iff* $a_s, a_o \in M$, $a_j \in T$. In the case of Spin, M is histories, while for Counter, it is $\mathbb{N}$. Neither Spin nor Counter requires shared ghost state, so T is the unit type for both[9]. We use $a_s\,(s)$, $a_j\,(s)$ and $a_o\,(s)$ as generic projections out of s, but introduce labels for readability; e.g., τ_s/τ_o and κ_s/κ_o for a_s/a_o in Spin and Counter, respectively.

The *self* and *other* components of different threads aren't independent, but offer thread-local views of the thread configuration. For example, imagine two threads θ_1 and θ_2 running in parallel. If the state of θ_1 is $s = (a_s, a_j, a_o)$, then the state of θ_2 *must be* the ***transposed*** $s^\top = (a_o, a_j, a_s)$. Both states represent the same configuration, with s being the view of θ_1 and $s^\top$ that of θ_2. The transposed states have swapped *self* and *other* components, thus capturing that θ_1 is the environment of θ_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 $(M, \bullet, \mathbb{1})$ where $\bullet$ (*join*) is a partial, commutative, associative, binary operation on M, with $\mathbb{1}$ 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 θ_1, θ_2 and θ_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 θ is the parent thread of θ_1 and θ_2, then its state is $s = (a_1 \bullet a_2, a_j, a_3)$, since θ is the combination of θ_1 and θ_2, and has θ_3 as the environment. We abbreviate as $s = s_1 * s_2$ the relationship between the parent state s, and the children states s_1 and s_2.

[9]Section 4 will show an example with a non-trivial T.

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 $s \lhd p = (a_s \bullet p, a_j, a_o)$. Dually, the* other-*framing of s with p is the state $s \rhd p = (a_s, a_j, p \bullet a_o)$.*

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 $P\,(s \rhd p) \leftrightarrow P\,(s \lhd p)$ 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 $f\,(s \rhd p) = f\,(s \lhd p)$ under the same conditions. Given a global P, the predicate Q is P-***global***, if $P(s \lhd p) \rightarrow Q(s \rhd p) \leftrightarrow Q(s \lhd p)$, and similarly for functions.

Examples of global predicates from Section 2, are $P\,s = \tau_s\,s \perp \tau_o\,s$, and $Q\,s = \mathsf{alternate}\,(\hat{\tau}\,s)$ used in Figure 2 to characterize Spin histories. P and Q are both defined in terms of $\hat{\tau}\,s = \tau_s\,s \bullet \tau_o\,s$; Q directly so, and P because $\tau_s\,s \perp \tau_o\,s$ iff $\tau_s\,s \bullet \tau_o\,s$ 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 τ_s and τ_o, but don't alter the total. In fact, $\hat{\tau}$ 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 $\Sigma = (P, \ulcorner - \urcorner)$, where P is a global predicate on (M, T)-state and $\ulcorner - \urcorner$ is a partial P-global function from (M, T)-states into heaps, called *erasure*, such that for every state s, $P\,s$ implies $a_s(s) \perp a_o(s)$ and $\ulcorner s \urcorner$ is defined. We write $s \in \Sigma$ to mean $P\,s$.

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.

$$(P * Q)\,s \;\widehat{=}\; \exists s_1, s_2.\, s = s_1 * s_2 \wedge P\,s_1 \wedge Q\,s_2$$

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

(1) *(partial function)* if $t\ s\ s'_1$ and $t\ s\ s'_2$ then $s'_1 = s'_2$.
(2) *(other-fixity)* if $t\ s\ s'$, then $a_o\,s = a_o\,s'$
(3) *(transition locality)* if $t\ (s \rhd p)\ x$, then there exists s' such that $x = s' \rhd p$ and $t\ (s \lhd p)\ (s' \lhd p)$

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 lock_tr in Figure 2 uses $\tau_o\,s$ to compute a fresh timestamp. To explain transition locality (3), let θ_1 be a thread in the state $s \rhd p$, whose sibling θ_2 has *self*-component p. Their parent θ is thus in the state $s \lhd p$. If θ_1 performs a transition $t\ (s \rhd p)\ x$, then by (3), the move can be seen as a transition of θ in the state $s \lhd p$. Intuitively, the parent is "held responsible" for the transitions of its children.

Transition locality requires that the frame p must be available in a_o, from which it is removed to be framed onto a_s. For example, we can't frame an arbitrary history p onto $\tau_s\, s$ in the transitions of Spin in Figure 2, but only a history p that can be taken out of $\tau_o\, s$. In particular, a transfer of p preserves the value of global functions, such as, e.g. $\hat{\tau}\, s$. 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 $\tau_s\, s$,

$$\begin{aligned}\mathsf{lock} : [k].\ & \{\lambda s.\, \tau_s\, s = \emptyset \wedge k \leq \mathsf{last\_stamp}\,(\tau_o\, s)\}\\ & \{\lambda s.\, \tau_s\, s = t \Mapsto \mathsf{L} \wedge k < t\}@\mathsf{Spin}\end{aligned}$$

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

$$\begin{aligned}\mathsf{lock} : [h, k].\ & \{\lambda s.\, \tau_s\, s = h \wedge k \leq \mathsf{last\_stamp}\,(\hat{\tau}\, s)\}\\ & \{\lambda s.\, \tau_s\, s = h \bullet t \Mapsto \mathsf{L} \wedge k < t\}@\mathsf{Spin}\end{aligned}$$

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

Definition 3.5. A Σ-transition t is ***footprint preserving*** if $t\ s\ s'$ implies that $\ulcorner s \urcorner$ and $\ulcorner s' \urcorner$ 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 be tensored 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 Σ is an (M, T) state space, and Δ a set of footprint preserving Σ 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 $a : \Sigma\,(V) \rightharpoonup \Delta\,(V) \times A$, mapping input state to output transition and value, which is ***local***, in the sense that it is invariant under framing. Formally, if $a\ (s \rhd p) = (t, v)$ then $a\ (s \lhd p) = (t, v)$; 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 $[a] : \Sigma\,(V) \rightharpoonup \Sigma\,(V) \times A$ mapping input state to output *state* and value, defined as $[a]\ s = (s', v)$ iff $\exists t.\, a\ s = (t, v) \wedge t\ s\ s'$. 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:

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

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 $[a]$ behaves like c when the states are erased to heaps. In other words, if $[a]\ s = (s', v)$, then $c \ulcorner s \urcorner = (\ulcorner s' \urcorner, v)$. 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 $r :=$ false.

As actions return transitions, we are justified in viewing an execution of a program over resource V as a path through the graph that is V, as mentioned in Section 1.

3.3 Morphisms

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

(1) *(locality of f_Σ)* there exists a function $\phi : M\,(W) \longrightarrow M\,(V)$ such that if $f_\Sigma\,(s_w \rhd p) = s_v$, then there exists s'_v such that $s_v = s'_v \rhd \phi\,(p)$, and $f_\Sigma\,(s_w \lhd p) = s'_v \lhd \phi\,(p)$.
(2) *(locality of f_Δ)* if $f_\Delta\,(s_w \rhd p)(t_v) = t_w$, then $f_\Delta\,(s_w \lhd p)(t_v) = t_w$.
(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 $f_\Delta\,(s_w)(t_v)$.

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_Δ 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 $f\ n$: Spin $\longrightarrow$ 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 ϕ in (1) with the first projection function, and it is easy to see that it satisfies the rest of (1). Property (2) holds because $(f\ n)_\Delta$ doesn't depend on the state argument, hence framing this state doesn't change the output. Finally, in (3), the values $a_o\,(s_w)$ and $a_o\,(s'_w)$ are also pairs of a history and a nat. If the pairs are equal, then their history components are equal too, deriving (3).

[10]All the actions we use in this paper and in the Coq code erase to some RMW command. However, we proved this only by hand, as the Coq implementation doesn't currently issue proof obligations to check this.

$$\begin{tikzcd}
s'_v & I\, s'_w \arrow[l, "f_\Sigma"'] \\
s_v \arrow[u, "t_v"] \arrow[ur, phantom, "\llcorner"] & I\, s_w \arrow[l, "f_\Sigma"] \arrow[u, "f_\Delta\, s_w\, t_v"']
\end{tikzcd}
\qquad
\begin{tikzcd}
s'_v & I\, s'_w \arrow[l, "f_\Sigma"'] \\
s_v \arrow[u, "V"', "*" near end] \arrow[ur, phantom, "\lrcorner"] & I\, s_w \arrow[l, "f_\Sigma"] \arrow[u, "W"', "*" near end]
\end{tikzcd}$$

Fig. 4. Conditions for I to be an f-simulation.

3.4 Simulations

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

Definition 3.9. Given a morphism $f : V \longrightarrow W$, an ***f*-simulation** is a predicate I on W-states such that:

(1) if $I\, s_w$, and $f_\Sigma\ (s_w) = s_v$, and $t_v\ s_v\ s'_v$, then there exist $t_w = f_\Delta\ s_w\ t_v$ and s'_w such that $I\, s'_w$ and $f_\Sigma\ (s'_w) = s'_v$, and $t_w\ s_w\ s'_w$.
(2) if $I\, s_w$, and $f_\Sigma\ (s_w)$ exists, and $s_w \xrightarrow[W]{}^{*} s'_w$, then $I\, s'_w$, and $f_\Sigma\ (s'_w)$ exists, and $f_\Sigma\ (s_w) \xrightarrow[V]{}^{*} f_\Sigma\ (s'_w)$. Here, the relation $s \xrightarrow[W]{} s'$ denotes that s ***other-steps*** by W to s', i.e., that there exists a transition $t \in \Delta\,(W)$ such that $t\ s^{\top}\ s'^{\top}$. The relation $\xrightarrow[W]{}^{*}$ is the reflexive-transitive closure of $\xrightarrow[W]{}$, 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 posses the following basic categorical structure.

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

$$\begin{aligned}
(g \circ f)_\Sigma\ s_w &= f_\Sigma\ (g_\Sigma\ s_w) \\
(g \circ f)_\Delta\ s_w\ t_u &= g_\Delta\ s_w\ (f_\Delta\ (g_\Sigma\ s_w)\ t_u)
\end{aligned}$$

Identity morphism $\mathrm{id} : V \longrightarrow V$ is defined by $\mathrm{id}_\Sigma\ s = s$ and $\mathrm{id}_\Delta\ s\ t = t$. Composition is associative, with unit id, where two morphisms are equal if their Σ and Δ 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.

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: $\mathsf{ST}\,V\,A$ and $[\Gamma].\,\{P\}\,A\,\{Q\}@V$. The first type circumscribes programs that respect the transitions of the resource V, and return a value of type A if they terminate (if $A = \mathsf{unit}$, we elide it from the type, as we did in Section 2). The second type is a subset of $\mathsf{ST}\,V\,A$, selecting only those programs that satisfy the precondition P and postcondition Q, under the context Γ of logical variables.

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

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

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 $T = \Pi_{x:A}.\,[\Gamma].\,\{P\}\,B\,\{Q\}@V$:

$$\mathsf{fix} : (T \to T) \to T$$

Here, T serves as a loop invariant; in $\mathsf{fix}\,(\lambda f.\,e)$ 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 $e : \mathsf{ST}\,V\,A$, $e_i : \mathsf{ST}\,V\,A_i$, a is a V-action, $f : V \to W$ is a morphism, I is an f-simulation, J is a V-invariant, and $s \in \Sigma\,(V)$.

vrf_post	$: (\forall v\;s.\,J\,s \to Q_1\;v\;s \to Q_2\;v\;s) \to J\,s \to \mathsf{vrf}\;e\;Q_1\;s \to \mathsf{vrf}\;e\;Q_2\;s$
vrf_ret	$: (Q\;v)^\bullet\;s \to \mathsf{vrf}\;(\mathsf{ret}\;v)\;Q\;s$
vrf_bnd	$: \mathsf{vrf}\;e_1\;(\lambda x.\,\mathsf{vrf}\;(e_2\;x)\;Q)\;s \to \mathsf{vrf}\;(x \leftarrow e_1; (e_2\;x))\;Q\;s$
vrf_par	$: (\mathsf{vrf}\;e_1\;Q_1) * (\mathsf{vrf}\;e_2\;Q_2)\;s \to \mathsf{vrf}\;(e_1 \parallel e_2)\;(\lambda v{:}A_1{\times}A_2.\,(Q_1\;v.1) * (Q_2\;v.2))\;s$
vrf_frame	$: (\mathsf{vrf}\;e\;Q_1) * Q_2^\bullet\;s \to \mathsf{vrf}\;e\;(\lambda v.\,(Q_1\;v) * Q_2)\;s$
vrf_act	$: (\lambda s'.\,\exists s''\;v.\,[a]\;s' = (s'', v) \wedge (Q\;v)^\bullet\;s'')^\bullet\;s \to \mathsf{vrf}\;\langle a\rangle\;Q\;s$
vrf_morph	$: f\hat{}(\mathsf{vrf}\;e\;Q)\;s \to I\,s \to \mathsf{vrf}\;(\mathsf{morph}\;f\;e)\;(\lambda v\;s'.\,f\hat{}(Q\;v)\;s' \wedge I\,s')\;s$ where $(f\hat{}P)\,(s) \mathrel{\widehat{=}} f_\Sigma\,(s)$ exists and $P\,(f_\Sigma\,(s))$

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 J 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 ***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]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.

predicate Q is $Q^\bullet\,(s) \;\widehat{=}\; \forall s'.\, s \xrightarrow[V]{}^{*} s' \rightarrow Q\,(s')$. Q is ***stable*** if $Q = Q^\bullet$, and it is easy to see that $Q^\bullet$ 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

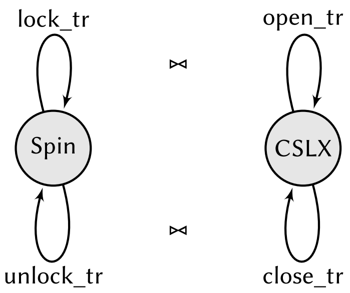
We next illustrate a more involved application of morphisms and simulation: how to derive a resource and methods for *exclusive* locking, à la CSL [O'Hearn 2007], from a resource for general locking (Section 2). An exclusive lock protects a shared heap, satisfying a user-supplied predicate R (*aka.* 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.

$$\begin{aligned}
\mathsf{exlock} : &\{\lambda s.\, \mu_s\, s = \overline{\mathsf{own}} \wedge \chi_s\, s = \emptyset\}\\
&\{\lambda s.\, \mu_s\, s = \mathsf{own} \wedge R\,(\chi_s\, s)\}@\mathsf{CSL}\\
\mathsf{exunlock} : &\{\lambda s.\, \mu_s\, s = \mathsf{own} \wedge R\,(\chi_s\, s)\}\\
&\{\lambda s.\, \mu_s\, s = \overline{\mathsf{own}} \wedge \chi_s\, s = \emptyset\}@\mathsf{CSL}
\end{aligned}$$

Here $\mu_s\, s$ is a ghost component of type $O = \{\mathsf{own}, \overline{\mathsf{own}}\}$, signifying whether "we" own the lock or not, and $\chi_s\, s$ is "our" private heap. O has PCM structure with the join defined by $x \bullet \overline{\mathsf{own}} = \overline{\mathsf{own}} \bullet x = x$, so that $\overline{\mathsf{own}}$ is the unit of the operation. We leave $\mathsf{own} \bullet \mathsf{own}$ 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, $\alpha_s\, s \in O$ keeps track of whether we own the lock or not, σ_s and σ_j are the private and shared heap respectively, and the transitions open_tr and close_tr move the heap from shared to private and back, respectively. We want to combine the Spin and CSLX resources as shown in the figure, by combining their state spaces, and coupling open_tr with lock_tr, and close_tr with unlock_tr, so that the transitions execute simultaneously. This will give us an intermediate resource CSL′, and a morphism f : Spin $\longrightarrow$ CSL′ defined similarly as in Section 2:

$$\begin{aligned}
f_\Sigma\, s &= s\backslash 1\\
f_\Delta\, s\ \mathsf{lock\_tr} &= \mathsf{lock\_tr} \bowtie \mathsf{open\_tr}\\
f_\Delta\, s\ \mathsf{unlock\_tr} &= \mathsf{unlock\_tr} \bowtie \mathsf{close\_tr}
\end{aligned}$$



State space: $s \in \Sigma\,(\text{Spin})$ iff
$(\tau_s\, s \perp \tau_o\, s) \wedge \text{alternate}\,(\hat{\tau}\, s) \wedge$
$r \neq \text{null}$
Erasure: $\ulcorner s \urcorner \mathrel{\widehat{=}} r \mapsto \omega\,(\hat{\tau}\, s)$
Transitions Δ(Spin):
$\text{lock\_tr}\ s\ s' \mathrel{\widehat{=}} \neg\omega\,(\hat{\tau}\, s) \wedge$
$\tau_s\, s' = \tau_s\, s \bullet \text{fresh}\,(\hat{\tau}\, s) \mapsto \text{L}$
$\pi_s\, s' = \pi_s\, s + 1$
$\text{unlock\_tr}\ s\ s' \mathrel{\widehat{=}} \omega\,(\hat{\tau} s) \wedge \pi_s\, s > 0 \wedge$
$\tau_s\, s' = \tau_s\, s \bullet \text{fresh}\,(\hat{\tau}\, s) \mapsto \text{U} \wedge$
$\pi_s\, s' = \pi_s\, s - 1$

State space $s \in \Sigma\,(\text{CSLX})$ iff
$\alpha_s\, s \perp \alpha_o\, s \wedge \sigma_s\, s \perp \sigma_j\, s \perp \sigma_o\, s \wedge$
$\text{if } \hat{\alpha}\, s = \text{own then } \sigma_j\, s = \emptyset \text{ else } R\,(\sigma_j\, s)$
Erasure: $\ulcorner s \urcorner \mathrel{\widehat{=}} \hat{\sigma}\, s \bullet \sigma_j\, s$
Transitions Δ (CSLX):
$\text{open\_tr}\ s\ s' \mathrel{\widehat{=}} \alpha_s\, s = \overline{\text{own}} \wedge \alpha_s\, s' = \text{own} \wedge$
$\sigma_s\, s' = \sigma_s\, s \bullet \sigma_j\, s \wedge \sigma_j\, s' = \emptyset$
$\text{close\_tr}\ s\ s' \mathrel{\widehat{=}} \alpha_s\, s = \text{own} \wedge \alpha_s\, s' = \overline{\text{own}} \wedge$
$\sigma_s\, s = \sigma_s\, s' \bullet \sigma_j\, s' \wedge R\,(\sigma_j\, s')$

Fig. 5. Redefinition of Spin, and CSLX resource for heap transfer in exclusive locking (in our implementation, Spin contains external transitions for receiving and giving away permissions to unlock, and CSLX contains transitions for reading and writing pointers in σ_s; we elide both for simplicity).

We will then restrict CSL′ into the CSL resource that we used in the specs for exlock and exunlock, as we shall describe. The components μ_s and χ_s used in these specs will be projections out of the state components of CSL.

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

The analysis implies that we should have defined Spin in a more general way, as shown in the left part of Figure 5. In particular, Spin should contain the integer components $\pi_s\,/\pi_o$ which indicate if unlock_tr is "enabled for us" ($\pi_s\, s > 0$), or not ($\pi_s\, s = 0$), and dually for others. These will give us the distinction we seek, as we shall see. In line with related work, we call π *permission to unlock*[12]. A thread may have more than one permission to unlock, which it can distribute among its children upon forking, who can then race to unlock. The addition of the new fields leads to the following minimal modification of the specs from Section 2, to indicate that lock enables unlock_tr, and a

[12]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[13].

$$\begin{aligned}
\mathsf{lock}' : [h,k].\ & \{\lambda s.\ \tau_s\, s = h \wedge k \leq \mathsf{last\_stamp}\,(\hat{\tau}\, s) \wedge \pi_s\, s = 0\} \\
& \{\lambda s.\ \exists t.\ \tau_s\, s = h \bullet t \Mapsto \mathsf{L} \wedge k < t \wedge \pi_s\, s = 1\}@\mathsf{Spin} \\
\mathsf{unlock}' : [h,k].\ & \{\lambda s.\ \tau_s\, s = h \wedge k \leq \mathsf{last\_stamp}\,(\hat{\tau}\, s) \wedge \pi_s\, s = 1\} \\
& \{\lambda s.\ \exists t.\ \tau_s\, s = h \bullet t \Mapsto \mathsf{U} \wedge k < t \wedge \pi_s\, s = 0\ \vee \\
& \quad \tau_s\, s = h \wedge \hat{\tau}\, s\ t = \mathsf{U} \wedge k \leq t \wedge \pi_s\, s = 1\}@\mathsf{Spin}
\end{aligned}$$

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, α 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 τ. Thus, once Spin and CSLX are combined, the two components must satisfy

$$\begin{aligned}
\omega\,(\tau_s\, s) &= (\alpha_s\, s == \mathsf{own}) \\
\omega\,(\tau_o\, s) &= (\alpha_o\, s == \mathsf{own})
\end{aligned} \tag{1}$$

as a basic coherence property[14]. Furthermore, we want to encode exclusive locking, so we must require that only the thread that holds the lock has the permission to unlock:

$$\begin{aligned}
\pi_s\, s &= (\mathsf{if}\ \alpha_s\, s = \mathsf{own}\ \mathsf{then}\ 1\ \mathsf{else}\ 0) \\
\pi_o\, s &= (\mathsf{if}\ \alpha_o\, s = \mathsf{own}\ \mathsf{then}\ 1\ \mathsf{else}\ 0)
\end{aligned} \tag{2}$$

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

$$\tau_s\, s \perp_\omega \tau_o\, s \tag{3}$$

where $h \perp_\omega k$ is defined as

$$\begin{aligned}
&(\mathsf{last\_op}\, h = \mathsf{L} \longrightarrow \mathsf{last\_stamp}\, k < \mathsf{last\_stamp}\,(h)) \wedge \\
&(\mathsf{last\_op}\, k = \mathsf{L} \longrightarrow \mathsf{last\_stamp}\, h < \mathsf{last\_stamp}\,(k)) \wedge h \perp k
\end{aligned}$$

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

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

[13]The specs for lock′ and unlock′ are small wrt. $\pi_s\, s$, but large wrt. $\tau_s\, s$.

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

[15]Requirement (3) restricts only the *last* timestamp in h and k, not all timestamps hereditarily. This suffices for our proof.

involves restricting the PCM of CSL′ by *Inv*. The mathematical underpinnings of such restrictions involve developing the notions of *sub-PCMs*, *PCM morphisms* and *compatibility relations*, which we defer to Appendix B. Here, it suffices to say that the construction leads to the situation summarized by the following diagram:

$$\mathsf{Spin} \xrightarrow{f} \mathsf{CSL}' \xrightarrow{\iota} \mathsf{CSL}$$

where morphism ι is defined by $\iota_\Sigma\ s = s$ and $\iota_\Delta\ s\ t = t$. Intuitively, CSL states are a subset of CSL′ states satisfying *Inv*, and ι_Σ is the injection from $\Sigma\,(\mathsf{CSL})$ to $\Sigma\,(\mathsf{CSL}')$.

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

$$\mathit{Sim}\ s \ \widehat{=}\ \text{if } \alpha_s\ s = \mathsf{own} \text{ then } R\,(\sigma_s\ s) \text{ else } \sigma_s\ s = \emptyset$$

Unlike *Inv*, *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, *Sim is* an $(\iota \circ f)$-simulation, as it satisfies the commuting diagrams from Figure 4. For example, when Spin executes lock_tr, then CSL sets $\alpha_s\ s = \mathsf{own}$ and acquires the shared heap, thus making the self heap satisfy R. When Spin executes unlock_tr, then CSL returns the shared heap, making the self heap empty. In other words, *Sim* describes the state of CSL immediately after locking, and immediately before unlocking, which suffices for the morphing of lock′ and unlock′. We only show the derivation for exunlock = morph $(\iota \circ f)$ unlock′, and refer to the Coq code [Nanevski et al. 2019] for the derivation of exlock.

```
1. $\{\alpha_s\ s = \mathsf{own} \wedge R\,(\sigma_s\ s)\}$
2. $\{\tau_s\ s = h \wedge \pi_s\ s = 1 \wedge \alpha_s\ s = \mathsf{own} \wedge R\,(\sigma_s\ s)\}$
3. $\{\tau_s\ s = h \wedge k = \mathsf{last\_stamp}\,(\hat{\tau}\ s) \wedge \pi_s\ s = 1 \wedge \mathit{Sim}\,s\}$
4. morph $(\iota \circ f)$ unlock′    // using simulation $\mathit{Sim}$
5. $\{(\tau_s\ s = h \bullet t \Mapsto \mathsf{U} \wedge k < t \wedge \pi_s\ s = 0\ \vee$
     $\tau_s\ s = h \wedge k \leq t \wedge \hat{\tau}\ s\ t = \mathsf{U} \wedge \pi_s\ s = 1) \wedge \mathit{Sim}\,s\}$
6. $\{\tau_s\ s = h \bullet t \Mapsto \mathsf{U} \wedge k < t \wedge \pi_s\ s = 0 \wedge \mathit{Sim}\,s\}$
7. $\{\alpha_s\ s = \overline{\mathsf{own}} \wedge \sigma_s\ s = \emptyset\}$
```

The key step is in line 6, where we must derive that the second disjunct in line 5 must be false; that is, no other thread could have raced us to unlock before us in line 4. We infer that the disjunct is false by reasoning about the histories $\tau_s\ s$ and $\tau_o\ s$ in line 5. From $\pi_s\ s = 1$, it must be $\omega\,(\tau_s\ s) = \mathsf{true}$, by the invariant *Inv* which holds throughout, as s is a CSL state. In particular, $\mathsf{last\_op}\,(\tau_s\ s) = \mathsf{L}$. By *Inv* again, $\tau_s\ s$ and $\tau_o\ s$ are exclusive histories, so $\mathsf{last\_stamp}\,(\tau_o\ s) < \mathsf{last\_stamp}\,(\tau_s\ s)$, and it must be $\mathsf{last\_stamp}\,(\tau_s\ s) = \mathsf{last\_stamp}\,(\hat{\tau}\ s) = k$, because $\mathsf{last\_stamp}\,(\hat{\tau}\ s)$ is the maximum of $\mathsf{last\_stamp}\,(\tau_s\ s)$ and $\mathsf{last\_stamp}\,(\tau_o\ s)$. But this contradicts that $\hat{\tau}\ s$ contains entry U at $t \geq k$. We now obtain the desired spec of 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 $f : V \xrightarrow{X} W$ introduces a type X of indices for f. The state component $f_\Sigma : X \to \Sigma(W) \rightharpoonup \Sigma(V)$ and the transition component $f_\Delta : X \to \Sigma(V) \to \Delta(V) \rightharpoonup \Delta(W)$ now allow input X, and satisfy a number of properties, listed in Appendix C, which reduce to Definition 3.8 when X is the unit type. Similarly, f-simulations must be indexed too, to be predicates over X and $\Sigma(W)$, satisfying a number of properties which reduce to Definition 3.9 when $X = \text{unit}$.

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

$$\frac{e : \{P\}\, A\, \{Q\}@V}{\mathsf{morph}\ f\ x\ e : \{\lambda s_w.\, (f\ x)\hat{}P\ s_w \wedge I\ x\ s_w\}\ A\ \{\lambda s_w.\, \exists y.\, (f\ y)\hat{}Q\ s_w \wedge I\ y\ s_w\}@W}$$

Here, $(f\ x)\hat{}P\,s_w$ is defined as $\exists s_v.\, s_v = f_\Sigma\, x\, s_w \wedge P\, s_v$.

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

$$\begin{aligned}\mathsf{push}(v) : [k].\ &\{\lambda s.\, \sigma_s\, s = \emptyset \wedge \tau_s\, s = \emptyset \wedge k \le \mathsf{last\_stamp}\,(\tau_o\, s)\}\\ &\{\lambda s.\, \sigma_s\, s = \emptyset \wedge \exists t\ vs.\, \tau_s\, s = t \Mapsto (vs, v :: vs) \wedge k < t\}@\mathsf{Stack}\end{aligned}$$

Here τ_s, and τ_o are histories of stack's operations, as in the case of Spin, and σ_s is the thread-private heap. The spec says that push starts with $\tau_s\, s = \emptyset$ (by framing, any history) and ends with $\tau_s\, s = t \Mapsto (vs, v :: vs)$ to indicate that a push of v indeed occurred, and *after* all the timestamps from the pre-state. The Stack resource also has joint components σ_j (the heap storing the stack's physical layout) and α_j (abstract contents of the stack as a mathematical sequence). The push method first allocates a new node in σ_s, then moves it to σ_j where it is enlinked to the top of the laid-out stack, after which push updates τ_s and α_j to reflect the addition of the node. We elide the full definition of Stack as it isn't essential for the idea; it suffices to know that predicate layout describes how α_j is laid out in σ_j (i.e., $\forall s \in \Sigma\,(\mathsf{Stack}).\ \mathsf{layout}\,(\alpha_j\, s)\,(\sigma_j\, s)$), and that the global history $\hat{\tau}\, s$ has no timestamp gaps (i.e., $\forall s \in \Sigma\,(\mathsf{Stack}).\ \mathsf{dom}\,(\hat{\tau}\, s) = \{1, \ldots, \mathsf{last\_stamp}\,(\hat{\tau}\, s)\}$).

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

$$\begin{aligned}e : &\{\lambda s.\, \sigma_s\, s = \emptyset \wedge \tau_s\, s = \emptyset\}\\ &\{\lambda s.\, \sigma_s\, s = \emptyset \wedge \exists t_1\ vs_1\ t_2\ vs_2.\, \tau_s\, s = t_1 \Mapsto (vs_1, a :: vs_1) \bullet t_2 \Mapsto (vs_2, b :: vs_2)\}@\mathsf{Stack}\end{aligned}$$

The specification reflects that e pushes a and b, to change the stack contents from vs_1 to $a :: vs_1$ at time t_1, and from vs_2 to $b :: vs_2$ 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., $t_2 = t_1 + 1 \vee t_1 = t_2 + 1$), 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 σ_s and σ_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][16]. A program working over Priv can install an empty stack in σ_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*

[16]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:

$$\{\lambda s.\, \mathsf{layout\ nil}\,(\sigma_s\, s)\}\ \{\lambda s.\, \mathsf{layout}\,[a,b]\,(\sigma_s\, s) \vee \mathsf{layout}\,[b,a]\,(\sigma_s\, s)\}@\mathsf{Priv}$$

The move from Stack to Priv thus essentially *forgets* the ghost state of histories, and the distinction in Stack between shared and private heaps. These components and distinctions are visible when in the scope of Stack, but hidden when in Priv[17]. We would like to obtain the Priv spec by applying the morph rule to the Stack spec of e, with a morphism $f : \mathsf{Stack} \longrightarrow \mathsf{Priv}$ that forgets the histories. Unfortunately, such a morphism can't be constructed as-is. Were it to exist, then f_Σ, being contravariant, should map a state s_{Priv}, containing only heaps, to a state s_{Stack}, containing heaps *and histories*; thus f_Σ must "invent" the history component out of thin air.

This is where families come in. We make $f : \mathsf{Stack} \overset{\mathsf{hist}}{\longrightarrow} \mathsf{Priv}$ a family over $X = \mathsf{hist}$, thereby passing to f_Σ the history x that should be added to s_{Priv} to produce an s_{Stack}.

$$\begin{aligned} f_\Sigma\ x\ s_{\mathsf{Priv}} = s_{\mathsf{Stack}} \mathrel{\widehat{=}} {} & \sigma_s\,(s_{\mathsf{Priv}}) = \sigma_s\,(s_{\mathsf{Stack}}) \bullet \sigma_j\,(s_{\mathsf{Stack}}) \wedge \\ & \sigma_o\,(s_{\mathsf{Priv}}) = \sigma_o\,(s_{\mathsf{Stack}}) \wedge \\ & \tau_s\,(s_{\mathsf{Stack}}) = x \wedge \tau_o\,(s_{\mathsf{Stack}}) = \emptyset \end{aligned}$$

The first conjunct directly states that Stack is installed in $\sigma_s\,(s_{\mathsf{Priv}})$ by making $\sigma_s\,(s_{\mathsf{Priv}})$ be the join of the heaps $\sigma_j\,(s_{\mathsf{Stack}})$ and $\sigma_s\,(s_{\mathsf{Stack}})$[18]. The second conjunct says that the heap $\sigma_o\,(s_{\mathsf{Priv}})$ of the interfering threads is propagated to $\sigma_o\,(s_{\mathsf{Stack}})$. The third conjunct captures that the history component of s_{Stack} is set to the index x, as discussed immediately above. In the last conjunct, the $\tau_o\,(s_{\mathsf{Stack}})$ history is declared $\emptyset$, thus formalizing quiescence. We elide the definition of f_Δ; it suffices to know that it maps a Stack transition (relation over heap and ghost state of Stack) to its "erasure", i.e. a relation over heaps of Priv representing a single-pointer operation such as read, write and CAS, but ignoring the ghost histories[19]. Finally, applying the new morph rule with f to the Stack specification of e, with $I\ x$ being the always-true predicate on Priv states, and $x = \emptyset$, gives us exactly the desired quiescent Priv spec. The full proof is in the Coq files, but briefly: let s_{Priv} be the state in the postcondition; then there exists s_{Stack} and a history y such that $\tau_s\,(s_{\mathsf{Stack}}) = y = t_1 \Mapsto (vs_1, a :: vs_1) \bullet t_2 \Mapsto (vs_2, b :: vs_2)$, and $\tau_o\,(s_{\mathsf{Stack}}) = \emptyset$, and $\sigma_s\,(s_{\mathsf{Priv}}) = \sigma_j\,(s_{\mathsf{Stack}})$. From $\Sigma\,(\mathsf{Stack})$, we know $\mathsf{layout}\,(\alpha_j\,(s_{\mathsf{Stack}}))\,(\sigma_j\,(s_{\mathsf{Stack}}))$, and hence $\mathsf{layout}\,(\alpha_j\,(s_{\mathsf{Stack}}))\,(\sigma_s\,(s_{\mathsf{Priv}}))$. Also from $\Sigma\,(\mathsf{Stack})$, we know that $\hat{\tau}\,(s_{\mathsf{Stack}})$ has no timestamp gaps; thus $\{t_1, t_2\} = \mathsf{dom}\,(\hat{\tau}\,(s_{\mathsf{Stack}})) = \{1, 2\}$, i.e., t_1 and t_2 are the only, and consecutive, events. But then $\alpha_j\,(s_{\mathsf{Stack}})$ must be either $[a, b]$ or $[b, a]$, giving us the desired postcondition in 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 $f : V \longrightarrow W$, f_Σ 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_Δ 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_{Stack} out of s_{Priv}, we have to identify the part of $\sigma_s\,(s_{\mathsf{Priv}})$ which we want to assign to $\sigma_j\,(s_{\mathsf{Stack}})$. This part has to be uniquely determined, else f_Σ 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 $x \longrightarrow y$ is a V-transition, then $f(x) \longrightarrow f(y)$ is a W-transition, and if $f(x) \longrightarrow t$ is a W-transition, then there exists y such that $x \longrightarrow y$ 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. $\mathsf{id\_tr}\ P = \lambda s\ s'.\, P\ s \wedge s' = s$. 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; Khyzha et al. 2017; Liang and Feng 2013; Schellhorn et al. 2012], the goal is to explicitly relate two *programs*, typically one concurrent, the other sequential. The sequential program then serves as a spec for the concurrent one, and can replace it in any larger context. Our goal in this paper is somewhat different; we seek to identify the concurrent program's *type*, which for us takes the form of a Hoare triple enriched with a resource. The type serves as the program's interface, and, as standard in type theory, any two programs with the same type can be interchanged in clients' code and proofs. As clients can already reason about the program via this type, the program *shouldn't need* a spec in the form of another program.

There are several advantages of 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 $l_1 \Mapsto \mathsf{Spin} \uplus l_2 \Mapsto \mathsf{Counter}$, 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

Martín 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 linearisability. 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.

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, Jérémie 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, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm 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 Incze, 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 Sieczkowski, 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)*. 639–667.

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).

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Conference of the European Associacion for Computer Science Logic (CSL)*. 1–19.

Susan S. Owicki and David Gries. 1976. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM* 19, 5 (1976), 279–285.

François Pottier. 2008. Hiding local state in direct style: a higher-order anti-frame rule. In *IEEE Symposium on Logic in Computer Science (LICS)*. 331–340.

J.J.M.M. Rutten. 2000. Universal coalgebra: a theory of systems. *Theoretical Computer Science* 249, 1 (2000), 3 – 80.

Gerhard Schellhorn, Heike Wehrheim, and John Derrick. 2012. How to prove algorithms linearisable. In *Computer Aided Verification (CAV)*. 243–259.

Jan Schwinghammer, Lars Birkedal, François Pottier, Bernhard Reus, Kristian Støvring, and Hongseok Yang. 2013. A step-indexed Kripke model of hidden state. *Mathematical Structures in Computer Science (MSCS)* 23, 1 (2013), 1–54.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and verifying concurrent algorithms with histories and subjectivity. In *European Symposium on Programming (ESOP)*. 333–358.

Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. 2016. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 92–110.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. In *ACM Symposium on Principles of Programming Languages (POPL)*. 28:1–28:30.

Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *European Symposium on Programming (ESOP)*. 149–168.

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular reasoning about separation of concurrent data structures. In *European Symposium on Programming (ESOP)*. 169–188.

The Coq Development Team. 2018. *The Coq Proof Assistant Reference Manual - Version V8.8.1.* http://coq.inria.fr/refman/.

R. Kent Treiber. 1986. *Systems programming: coping with parallelism.* Technical Report RJ 5118. IBM Almaden Research Center.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ACM International Conference on Functional Programming (ICFP)*. 377–390.

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, $\mathbb{1}_{A \times B} = (\mathbb{1}_A, \mathbb{1}_B)$.*

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

Definition A.2. Let $s_i = (a^i_s, a^i_j, a^i_o)$ 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 $[s_1, s_2] \mathrel{\widehat{=}} ((a^1_s, a^2_s), (a^1_j, a^2_j), (a^1_o, a^2_o))$, and projection states of s are $s\backslash i \mathrel{\widehat{=}} (a_s\backslash i, a_j\backslash i, a_o\backslash i)$, $i = 1, 2$. The usual beta and eta laws for products hold, i.e.: $[s_1, s_2]\backslash i = s_i$ and $s = [s\backslash 1, s\backslash 2]$.

Definition A.3. Given state spaces Σ_1 and Σ_2, the ***product state space*** $\Sigma_1 \times \Sigma_2$ is defined by:

$$\begin{array}{rcl} \ulcorner s \urcorner & \mathrel{\widehat{=}} & \ulcorner s\backslash 1 \urcorner^{\Sigma_1} \bullet \ulcorner s\backslash 2 \urcorner^{\Sigma_2} \\ s \in \Sigma_1 \times \Sigma_2 & \text{iff} & s\backslash 1 \in \Sigma_1 \wedge s\backslash 2 \in \Sigma_2 \wedge \mathsf{defined}\, \ulcorner s \urcorner. \end{array}$$

Definition A.4. Let t_i be a Σ_i-transition, $i = 1, 2$. The ***coupling of*** t_1 ***and*** t_2 is the $\Sigma_1 \times \Sigma_2$-transition defined as $(t_1 \bowtie t_2)\ s\ s' \mathrel{\widehat{=}} t_1\ (s\backslash 1)\ (s'\backslash 1) \wedge t_2\ (s\backslash 2)\ (s'\backslash 2)$ [20].

The coupling of transitions is used extensively in our constructions, e.g. in $\mathsf{lock\_tr} \bowtie \mathsf{open\_tr}$ 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: $\mathsf{lock\_tr}$ from Spin and $\mathsf{open\_tr}$ from $\mathsf{Counter}$ 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 $t_v \bowtie t_w$ 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:

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

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

$$I = \{(\mathsf{lock\_tr}, \mathsf{open\_tr}),\ (\mathsf{unlock\_tr}, \mathsf{close\_tr}),\ (\mathsf{id\_tr}, \mathsf{id\_tr})\}$$

Here, id is the idle transition defined as $\lambda s\ s'.\, s' = s$. 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.

[20] In 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 $\ulcorner s' \urcorner$ 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

$$\begin{aligned} x \perp_\omega y = \; & (\mathsf{last\_op}\, x = \mathsf{L} \longrightarrow \mathsf{last\_stamp}\, y < \mathsf{last\_stamp}\,(x)) \wedge \\ & (\mathsf{last\_op}\, y = \mathsf{L} \longrightarrow \mathsf{last\_stamp}\, x < \mathsf{last\_stamp}\,(y)) \wedge x \perp y \end{aligned}$$

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 $\mathsf{last\_op}\, x = \mathsf{L}$. 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 $\mathsf{max}(t_y, t_z) \leq t_y, t_z < t_x$.

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

Now we can define PCM morphisms.

Definition B.2. A ***PCM morphism*** $\phi : A \longrightarrow B$ 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 ϕ 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 π, τ, μ, σ, etc., are all PCM morphisms. For example, τ and π in Figure 5 are projections out of the PCM $M\,(\mathsf{Spin})$ 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 π.

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 $\iota : A \longrightarrow B$ (injection) and a morphism $\rho : B \longrightarrow A$ (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 ι is injective, i.e., if we coerce $a \in A$ into $\iota\, a$, we can recover a back by ρ, since no other element of A maps by ι into $\iota\, a$. The dual property (2) allows the same for a subset of B's elements, that are related by $\perp_\rho$ to $\mathbb{1}_B$. 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 ρ, 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 $M\,(\mathsf{CSL}')$.

Lemma B.4. *Given PCM A and compatibility relation R on A, the set $A/R = \{x \in A \mid x\, R\, \mathbb{1}_A\}$ forms a PCM under unit $\mathbb{1}_A$, 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 ι and retraction ρ defined as $\forall x \in A/R.\, \iota\,(x) = x$ and $\forall a \in A.\, \rho\,(a) = a$ if $a\, R\, \mathbb{1}_A$, 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 $M\,(\mathsf{CSL}')/Inv$.

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 $s \in \Sigma\,(V) \wedge (a_s\, s)\, R\, (a_o\, s)$

There is a generic resource morphism $\iota : V \longrightarrow V/R$ 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 Inv, and the resource morphism $\iota : \mathsf{CSL}' \longrightarrow \mathsf{CSL}$ 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.

$$\begin{array}{llll} (f \circ g)\,(x) \;\widehat{=}\; f\,(g\,x) & & x \perp_{f \circ g} y \;\widehat{=}\; x \perp_g y \wedge g\,x \perp_f g\,y \\ (f \bullet g)\,(x) \;\widehat{=}\; f\,x \bullet g\,x & & x \perp_{f \bullet g} y \;\widehat{=}\; x \perp_f y \wedge x \perp_g y \wedge f\,(x \bullet y) \perp g\,(x \bullet y) \end{array}$$

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.

$$\begin{array}{rl} x\,(\ker f)\,y & \widehat{=}\; x \perp_f y \wedge f\,x = f\,y = \mathbb{1} \\ x\,(\mathrm{eql}\; f\; g)\,y & \widehat{=}\; x \perp_f y \wedge x \perp_g y \wedge f\,x = g\,x \wedge f\,y = g\,y \end{array}$$

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.

$$(f/R)\,(x) \;\widehat{=}\; \begin{cases} f\,x, & \text{if } x\, R\, \mathbb{1} \\ \text{undefined}, & \text{otherwise} \end{cases} \qquad \text{with} \quad x \perp_{f/R} y \;\widehat{=}\; x \perp_f y \wedge x\, R\, y$$

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 $\omega' : \mathsf{hist} \longrightarrow O$ is the morphism defined on a history h as $\omega'\, h = \mathsf{if}\ \omega\, h$ then own else $\overline{\mathsf{own}}$.

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*** $f : V \overset{X}{\longrightarrow} W$ (or just *family*), consists of partial functions $f_\Sigma : X \longrightarrow \Sigma(W) \rightharpoonup \Sigma(V)$ (note the contravariance), and $f_\Delta : X \longrightarrow \Delta(V) \rightharpoonup \Delta(W)$ on transitions, such that:

1. (locality of f_Σ) there exits a function $f_A : M(W) \longrightarrow M(V)$ such that if $f_\Sigma\ x\,(s_w \rhd p) = s_v$, then $s_v = s'_v \rhd f_A\,(p)$, and $f_\Sigma\ x\,(s_w \lhd p) = s'_v \lhd f_A\,(p)$.
2. (locality of f_Δ) if $f_\Delta\, x\,(s_w \rhd p)\, t = t'$, then $f_\Delta\, x\,(s_w \lhd p)\, t = t'$.
3. (other-fixity) if $a_o\,(s_w) = a_o\,(s'_w)$ and $f_\Sigma\, x\, s_w$, $f_\Sigma\, x'\, s'_w$ exist, then $a_o\,(f_\Sigma\, x\, s_w) = a_o\,(f_\Sigma\, x' s'_w\,)$.
4. (injectivity on indices) if $f_\Sigma\, x_1\, s_{w_1}$ and $f_\Sigma\, x_2\, s_{w_2}$ exist and $f_\Sigma\, x_1\, s_{w_1} = f_\Sigma\, x_2\, s_{w_2}$, 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 $f_\Sigma\, x\, s_w$ uniquely determines the index x. In the Stack example in Section 5, it is easy to see that the definition of f_Σ satisfies this property, because equal states have equal histories.

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

1. if $I\, x\, s_w$, and $f_\Sigma\ x\, s_w = s_v$, and $t\ s_v\ s'_v$, then there exist x', $t' = f_\Delta\ x\ s_w\ t$, and s'_w such that $I\, x'\, s'_w$ and $f_\Sigma\ x'\, s'_w = s'_v$, and $t'\ s_w\ s'_w$.
2. if $I\, x\, s_w$, and $s_v = f_\Sigma\, x\, s_w$ exists, and $s_w \underset{W}{\longrightarrow}^* s'_w$, then $I\, x\, s'_w$, and $s'_v = f_\Sigma\, x\, s'_w$ exists, and $s_v \underset{V}{\longrightarrow}^* s'_v$.

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 $x = \mathsf{empty}$ (i.e., write morph empty f e), then this index will evolve with e taking the transitions of Stack to track how e changes the self history by adding the entries for pushing 1 and 2. In property (2), the index x simply propagates.

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 ST $V\ A$ type and the Hoare triple type will be *complete lattices* of sets of trees, giving us a suitable setting for modeling recursion. The *soundness* of FCSL follows from showing that the lemmas about the vrf predicate transformer listed in Section 3, are satisfied by the denotations of the commands.

We choose the Calculus of Inductive Constructions (CɪC) [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 CɪC 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 λ-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 CɪC.

D.1 Action trees and program denotations

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

$$\begin{aligned}
\mathsf{tree}\ V\ A\ \mathrel{\widehat{=}}\ & \mathsf{Unfinished}\\
|\ & \mathsf{Ret}\ (v : A)\\
|\ & \mathsf{Act}\ (a : \mathsf{action}\ V\ A)\\
|\ & \mathsf{Seq}\ (T : \mathsf{tree}\ V\ B)\ (K : B \longrightarrow \mathsf{tree}\ V\ A)\\
|\ & \mathsf{Par}\ (T_1 : \mathsf{tree}\ V\ B_1)\ (T_2 : \mathsf{tree}\ V\ B_2)\\
& \quad (K : B_1 \times B_2 \longrightarrow \mathsf{tree}\ V\ A)\\
|\ & \mathsf{Morph}\ (f : W \overset{X}{\to} V)\ (x : X)\ (T : \mathsf{tree}\ W\ A)
\end{aligned}$$

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 : action $V\ A$, 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. CɪC's iterated inductive definition permits the recursive occurrences of tree to be *nonuniform* (e.g., tree B_i in Par) and *nested* (e.g., the *positive* occurrence of tree A in the continuation). Since the CɪC function space includes case-analysis, the continuation may branch upon the argument. The Morph constructor embeds an index $x : X$, morphism $f : W \overset{X}{\to} V$, and tree T : tree $W\ A$ for the underlying computation. The constructor will denote T should be executed so that each of its actions is modified by f with an index x. We can now define the denotational model of our programs; that is the type ST $V\ A$ of sets of trees, containing Unfinished.

$$\mathsf{ST}\ V\ A\ \mathrel{\widehat{=}}\ \{e : \mathsf{set}\ (\mathsf{tree}\ V\ A) \mid \mathsf{Unfinished} \in e\}$$

$$\frac{}{\Delta \vdash \bar{x}, s, \mathsf{Seq}\ (\mathsf{Ret}\ v)\ K \xrightarrow{\mathsf{SeqRet}} \bar{x}, s, K\ v} \qquad \frac{\Delta \vdash \bar{x}, s, T \xrightarrow{\pi} \bar{x}, s', T'}{\Delta \vdash \bar{x}, s, \mathsf{Seq}\ T\ K \xrightarrow{\mathsf{SeqStep}\ \pi} \bar{x}', s', \mathsf{Seq}\ T'\ K}$$

$$\frac{}{\Delta \vdash \bar{x}, s, \mathsf{Par}\ (\mathsf{Ret}\ v_1)\ (\mathsf{Ret}\ v_2)\ K \xrightarrow{\mathsf{ParRet}} \bar{x}, s, K\ (v_1, v_2)} \qquad \frac{\Delta \vdash \bar{x}, s, T_1 \xrightarrow{\pi} \bar{x}', s', T_1'}{\Delta \vdash \bar{x}, s, \mathsf{Par}\ T_1\ T_2\ K \xrightarrow{\mathsf{ParL}\ \pi} \bar{x}', s', \mathsf{Par}\ T_1'\ T_2\ K}$$

$$\frac{\Delta \vdash \bar{x}, s, T_2 \xrightarrow{\pi} \bar{x}', s', T_2'}{\Delta \vdash \bar{x}, s, \mathsf{Par}\ T_1\ T_2\ K \xrightarrow{\mathsf{ParR}\ \pi} \bar{x}', s', \mathsf{Par}\ T_1\ T_2'\ K} \qquad \frac{\mathsf{unwind\_act}\ \Delta\ \bar{x}\ \bar{x}'\ a\ s\ s'\ v}{\Delta \vdash \bar{x}, s, \mathsf{Act}\ a \xrightarrow{\mathsf{ChoiceAct}} \bar{x}', s'; \mathsf{Ret}\ v}$$

$$\frac{}{\Delta \vdash \bar{x}, s, \mathsf{Morph}\ f\ y\ (\mathsf{Ret}\ v) \xrightarrow{\mathsf{MorphRet}} \bar{x}, s, \mathsf{Ret}\ v}$$

$$\frac{\Delta, f \vdash (\bar{x}, y), s, T \xrightarrow{\pi} (\bar{x}', y'), s', T'}{\Delta \vdash \bar{x}, s, \mathsf{Morph}\ f\ y\ T \xrightarrow{\mathsf{MorphStep}\ \pi} \bar{x}', s', \mathsf{Morph}\ f\ y'\ T'}$$

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

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

$$\begin{array}{rcl}
\mathsf{ret}\ (r : A) & \mathrel{\widehat{=}} & \{\mathsf{Unfinished}, \mathsf{Ret}\ r\} \\
x \leftarrow e_1; e_2 & \mathrel{\widehat{=}} & \{\mathsf{Unfinished}\} \cup \\
& & \quad \{\mathsf{Seq}\ T_1\ K \mid T_1 \in e_1 \wedge \forall x. K\ x \in e_2\} \\
e_1 \parallel e_2 & \mathrel{\widehat{=}} & \{\mathsf{Unfinished}\} \cup \\
& & \quad \{\mathsf{Par}\ T_1\ T_2\ \mathsf{Ret} \mid T_1 \in e_1 \wedge T_2 \in e_2\} \\
\langle a \rangle & \mathrel{\widehat{=}} & \{\mathsf{Unfinished}, \mathsf{Act}\ a\} \\
\mathsf{morph}\ f\ x\ e & \mathrel{\widehat{=}} & \{\mathsf{Unfinished}\} \cup \{\mathsf{Morph}\ f\ x\ T \mid T \in e\}
\end{array}$$

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

The denotations of composed programs motivate why we denote programs by non-empty sets, i.e., why each denotation contains at least Unfinished. If we had a program Empty whose denotation is the empty set, then the denotation of $x \leftarrow \mathsf{Empty}; e'$, $\mathsf{Empty} \parallel e'$ and $\mathsf{morph}\ f\ x\ \mathsf{Empty}$ will all also be empty, thus ignoring that the composed programs exhibit more behaviors. For example, the parallel composition $\mathsf{Empty} \parallel e'$ 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 $\{\mathsf{Unfinished}\} \parallel e'$ is denoted by the set:

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

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

D.2 Operational semantics of action trees

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

First, the component Δ is a morphism context. This is a sequence, potentially empty, of morphism families

$$f_0 : V_1 \overset{X_0}{\rightarrow} W, f_1 : V_2 \overset{X_1}{\rightarrow} V_1, \dots, f_n : V \overset{X_n}{\rightarrow} V_n$$

We say that Δ has resource type $V \longrightarrow W$, and index type $(X_0, \cdots, X_n)$. An empty context $\cdot$ has resource type $V \longrightarrow V$ for any V.

Second, the components $\bar{x}$ and $\bar{x}'$ are tuples, of type $(X_0, \cdots, X_n)$, 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

$$\mathsf{morph}\ f_0\ x_0\ (\cdots(\mathsf{morph}\ f_n\ x_n\ e)\cdots),$$

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

Third, the components s_w and s'_w are W-states, and T_v, T'_v : tree V A, for some type A. The meaning of the judgment is that a tree T_v, when executed in a state s_w, under the context of morphisms Δ 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 Δ 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 Δ 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 π is of path type, identifying the position in the tree where we want to make a reduction.

$$\begin{array}{lllll}
\mathsf{path} & \widehat{=} & \mathsf{ChoiceAct} \mid \mathsf{SeqRet} & & \mid \mathsf{SeqStep}\ (\pi : \mathsf{path}) \mid \\
& & \mathsf{ParRet} & \mid \mathsf{ParL}\ (\pi : \mathsf{path}) & \mid \mathsf{ParR}\ (\pi : \mathsf{path}) \quad \mid \\
& & \mathsf{MorphRet} & \mid \mathsf{MorphStep}\ (\pi : \mathsf{path}) &
\end{array}$$

The key are the constructors ParL π and ParR π. In a tree which is a Par tree, these constructors identify that we want to reduce in the left and right subtree, respectively, iteratively following the path π. If the tree is not a Par tree, then ParL and 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, ChoiceAct identifies the head position in the tree of the form Act(a), SeqRet identifies the head position in the tree of the form Seq (Ret v) K (i.e., it identifies a position of a beta-reduction), SeqStep π identifies a position in the tree Seq T K, if π identifies a position within T, etc. We do not paths for trees of the form Unfinished and 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 ComSquares $\Delta\ \bar{x}\ \bar{x}'\ s_v\ s'_v\ t_v\ s_w\ s'_w\ t_w$ 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,…, f_n, in the context $\Delta = f_0; f_1; \cdots; f_n$).

$$\begin{tikzcd}
s'_v & s'_n \arrow[l, "f_{n\Sigma}\ x'_n"'] \arrow[r, phantom, "\cdots\cdots"] & s'_1 & s'_w \arrow[l, "f_{0\Sigma}\ x'_0"'] \arrow[dl, phantom, "\llcorner", very near start] \\
s_v \arrow[u, "t_v"] & s_n \arrow[l, "f_{n\Sigma}\ x_n"] \arrow[u, dashed, "t_n"] \arrow[r, phantom, "\cdots\cdots"] & s_1 \arrow[u, dashed, "t_1"] & s_w \arrow[l, "f_{0\Sigma}\ x_0"] \arrow[u, "t_w = f_{0\Delta}\ x_0\ s_w\ t_1"']
\end{tikzcd}$$

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

$$\begin{aligned}
&\mathsf{ComSquares}\ \cdot\ ()\ ()\ s_v\ s'_v\ t_v\ s_w\ s'_w\ t_w\ \widehat{=}\\
&\quad s_v = s_w \wedge s'_v = s'_w \wedge t_v = t_w\\
&\mathsf{ComSquares}\ (f_0 : V_1 \overset{X_0}{\longrightarrow} W), \Delta)\ (x_0, \bar{x})\ (x'_0, \bar{x}')\\
&\qquad\qquad s_v\ s'_v\ t_v\ s_w\ s'_w\ t_w\ \widehat{=}\\
&\quad \exists s_1\ s'_1\ t_1.\, \mathsf{ComSquares}\ \Delta\ \bar{x}\ \bar{x}'\ s_v\ s'_v\ t_v\ s_1\ s'_1\ t_1 \wedge\\
&\quad f_{0\Sigma}\ x_0\ s_w = s_1 \wedge f_{0\Sigma}\ x'_0\ s'_w = s'_1
\end{aligned}$$

We will often abbreviate ComSquares, and write:

$$\begin{aligned}
&\mathsf{unwind\_tr}\ \Delta\ \bar{x}\ \bar{x}'\ t_v\ s_w\ s'_w\ \widehat{=}\\
&\quad \exists s_v\ s'_v\ t_w.\, \mathsf{ComSquares}\ \Delta\ \bar{x}\ \bar{x}'\ s_v\ s'_v\ t_v\ s_w\ s'_w\ t_w\\
&\mathsf{unwind\_act}\ \Delta\ \bar{x}\ \bar{x}'\ a\ s_w\ s'_w\ v\ \widehat{=}\\
&\quad \exists s_v\ s'_v\ t_v\ t_w.\, a\ s_v = (t_v, v) \wedge\\
&\qquad\qquad\qquad \mathsf{ComSquares}\ \Delta\ \bar{x}\ \bar{x}'\ s_v\ s'_v\ t_v\ s_w\ s'_w\ t_w
\end{aligned}$$

Given a context Δ, input indices $\bar{x}$, input state s_w and input transition t_v, 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 Δ. The abbreviation unwind_act does the same, but for input action $a : \mathsf{action} V A$. In the process, unwind_act also computes the return value v.

In the frequent cases where Δ 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 $s, T \xrightarrow{\pi} s', T'$.

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 π does not actually determine an action or a redex in the tree T. For example, we may have $T = \mathsf{Unfinished}$ and $\pi = \mathsf{ParR}$. 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 π 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 : tree V A and π be a path. Then the predicate good T π recognizes ***good*** paths for a tree T as follows:

good (Act a)	ChoiceAct	$\widehat{=}$ true
good (Seq (Ret v) _)	SeqRet	$\widehat{=}$ true
good (Seq T _)	SeqRet π	$\widehat{=}$ good T π
good (Par (Ret _) (Ret _) _)	ParRet	$\widehat{=}$ true
good (Par T_1 T_2 _)	ParL π	$\widehat{=}$ good T_1 π
good (Par T_1 T_2 _)	ParR π	$\widehat{=}$ good T_2 π
good (Morph f x (Ret _))	MorphRet	$\widehat{=}$ true
good (Morph f x T)	MorphStep π	$\widehat{=}$ good T π
good T	π	$\widehat{=}$ false otherwise

Definition D.3 (Safe configurations). We say that a state s is ***safe*** for the tree T and path π, written $s \in \mathsf{safe}\ t\ \pi$ if:

$$\mathsf{good}\ T\ \pi \longrightarrow \exists s'\ T'.\ s, T \xrightarrow{\pi} s', T'$$

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

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

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

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

Lemma D.5 (Other-fixity of stepping). *Let $\Delta : V \longrightarrow W$ and $\Delta \vdash \bar{x}, s, T \xrightarrow{\pi} \bar{x}', s', T'$. Then $a_o(s) = a_o(s')$.*

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

Lemma D.7 (Stability of stepping). *Let $\Delta : V \longrightarrow W$ and $\Delta \vdash \bar{x}, s, T \xrightarrow{\pi} \bar{x}', s', T'$. Then $s^{\top} \underset{W}{\longrightarrow^*} s'^{\top}$.*

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

Lemma D.9 (Locality of stepping). *Let $\Delta : V \longrightarrow W$ and $\Delta \vdash \bar{x}, (s \rhd p), T \xrightarrow{\pi} \bar{x}', s', T'$. Then there exists s'' such that $s' = s'' \rhd p$, and $\Delta \vdash \bar{x}, (s \lhd p), T \xrightarrow{\pi} \bar{x}', (s'' \lhd p), T'$.*

Lemma D.10 (Safety-monotonicity of stepping). *If $s \rhd p \in \mathsf{safe}\ T\ \pi$ then $s \lhd p \in \mathsf{safe}\ T\ \pi$.*

Lemma D.11 (Frameability of stepping). *Let $s \rhd p \in \mathsf{safe}\ T\ \pi$, and $s \lhd p, T \xrightarrow{\pi} s', T'$. Then there exists s'' such that $s' = s'' \lhd p$ and $s \rhd p, T \xrightarrow{\pi}, s'' \rhd p, T'$.*

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 $f : V \xrightarrow{X} W$.

Lemma D.12 (Stepping under morphism). *Let $f : V \longrightarrow WX$ and $f\ x\ s_w = s_v$. Then the following hold:*

(1) if $s_v, T \xrightarrow{\pi} s'_v, T'$, then $\exists x'\ s'_w.\ f\ x'\ s'_w = s'_v$ and $f \vdash (x), s_w, T \xrightarrow{\pi} (x'), s'_w, T'$.
(2) if $f \vdash (x), s_w, T \xrightarrow{\pi} (x'), s'_w, T'$, then $\exists x'\ s'_v.\ f\ x'\ s'_w = s'_v$ and $s_v, T \xrightarrow{\pi} s'_v, T'$.

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

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 : tree V A, and ζ 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:

$$\begin{array}{l}
\mathsf{always}^{\zeta}\ T\ X\ s\ \mathrel{\widehat{=}} \\
\quad \mathsf{if}\ \zeta = \pi :: \zeta'\ \mathsf{then} \\
\qquad \forall s_2.\, (s \underset{V}{\longrightarrow}^{*} s_2) \to \mathsf{safe}\ T\ \pi\ s_2 \wedge X\ s_2\ T\ \wedge \\
\qquad\quad \forall s_3\ T'.\, (s_2, T \xrightarrow{\pi} s_3, T') \to \mathsf{always}^{\zeta'}\ T_2\ X\ s_3 \\
\quad \mathsf{else}\ \forall s_2.\, (s \underset{V}{\longrightarrow}^{*} s_2) \to X\ s_2\ T \\
\mathsf{always}\ T\ X\ s\ \mathrel{\widehat{=}}\ \forall \zeta.\, \mathsf{always}^{\zeta}\ T\ X\ s \\
\\
\mathsf{after}\ T\ Q\ \mathrel{\widehat{=}}\ \mathsf{always}\ T\ (\lambda\, s'\ T'.\, \forall v.\, T' = \mathsf{Ret}\ v \to Q\ v\ s')
\end{array}$$

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 ζ. 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.

$$\mathsf{vrf}\ e\ Q\ s\ \mathrel{\widehat{=}}\ \forall T \in e.\, \mathsf{after}\ T\ Q\ s$$

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

$$\forall s \in \Sigma(V).\, P\ s \to \mathsf{vrf}\ e\ Q\ s$$

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* $\{\mathsf{Unfinished}\}$ *as the bottom. Thus one can compute the least fixed point of every monotone function by Knaster-Tarski theorem.*
- *All program constructors are monotone.*