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 de Paris, 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, using an associated notion of morphism-specific *simulation*. We apply morphisms and simulations to programs verified under one resource, to compositionally adapt them to operate under another resource, thus facilitating proof reuse.

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

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

1 INTRODUCTION

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

On the other hand, composition is also the cornerstone of *type theory*, where types serve as the interface that abstracts the internal properties of programs and proofs. Because both are focused on

Authors' addresses: Aleksandar Nanevski, IMDEA Software Institute, Spain, aleks.nanevski@imdea.org; Anindya Banerjee, IMDEA Software Institute, Spain, anindya.banerjee@imdea.org; Germán Andrés Delbianco, IRIF–Université de Paris, France, gad@irif.fr; Ignacio Fábregas, IMDEA Software Institute, Spain, ignacio.fabregas@imdea.org.

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

$$e : \{P\}\{Q\}$$

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

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

1.1 Resources

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

$$e : \{P\}\{Q\} @ V$$

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

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

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

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

---

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

[2]The *Compare-and-Set* variant of CAS(r, a, b) [Herlihy and Shavit 2008] atomically sets the pointer r to b if r contains a, otherwise leaves r unchanged. It moreover returns a Boolean value denoting the success or failure of the operation.
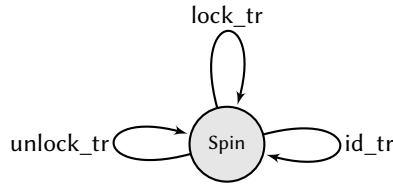
Fig. 1. Resource for spin locks. By convention, the *idle* transition id_tr will be elided in the future diagrams.

Figure 1 shows (an abstracted form of) the resource Spin, suitable to type lock.[3] Every execution of lock describes a path through Spin consisting of several idle transitions id_tr, corresponding to unsuccessful CAS's, followed by a locking transition lock_tr corresponding to a successful CAS. Similarly, the Spin resource also types the unlock program

$$\text{unlock} \; \widehat{=} \; r := \text{false}$$

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

1.2 Morphisms

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

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

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

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

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

---

[3]We'll define the state space and the transitions of Spin in Section 2, and eventually tie them to the implementations of lock and unlock. At this point, it suffices to know that a program in Spin may transition by lock_tr only if r is free (thereby locking it), and by unlock_tr only if r is locked (thereby freeing it).
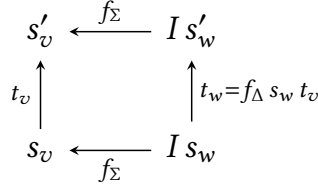
$$
\begin{array}{ccc}
s'_v & \xleftarrow{\quad f_\Sigma \quad} & I\,s'_w \\[4pt]
t_v \Big\uparrow & & \Big\uparrow t_w = f_\Delta\,s_w\,t_v \\[4pt]
s_v & \xleftarrow[\quad f_\Sigma \quad]{} & I\,s_w
\end{array}
$$

Fig. 2. Reinterpreting the transitions of a resource V into those of W, by a morphism $f : V \to W$ and an f-simulation I. In this diagram, and in the sequel, $I\,s$ denotes "state s such that the predicate I holds".

and natural transformations, etc. Morphisms endow the structure with dynamics and allow studying it under change. This will be the case for us as well.

More specifically, and akin to how automata homomorphisms [Ginzburg 1968] are defined componentwise, a resource morphism $f : V \to W$ consists of two partial functions f_Σ and f_Δ, acting respectively on *states* (Σ) and *transitions* (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 Figure 2, the morphed program inhabits W, so we describe it starting with a W-state s_w in the lower-right corner of the diagram (the predicate I that is applied to s_w in the diagram will be explained promptly). To compute the next state of the morphed program, we first take $s_v = f_\Sigma\,s_w$ which is a state in V (utilizing contravariance of f_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.

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

1.3 Simulations and Inference

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

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

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

$$
\text{where}\quad f\hat{}R\,s_w \ \widehat{=}\ \exists s_v.\ s_v = f_\Sigma\,s_w \wedge R\,s_v
$$

The Morph rule translates the requirements for a morphism $f : V \to W$, and a simulation I, expressed diagrammatically in Figure 2, from properties of states and transitions of the resources V and W, into the specification of a program morph $f\ e$ in W. The latter spec is structured, both in its pre- and the postconditions, as a conjunction of (i) the transformation of e's spec from V to W,

and (ii) a statement that the simulation I, being in fact an invariant of the resources' transitions, holds at the boundaries of the *morphed* execution of e. For the first part, we use the predicate $f\hat{\ }\_$, to lift the pre- and postconditions of e from states in V to states of W, which are related via f_Σ. This motivates the contravariance of f_Σ: we have verified e in the context of the resource V, but we intend to *morph e* and execute it in a new resource context, where states are inhabitants of W.

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

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

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

Moreover, resource morphisms go beyond mere program specification and proof, as they also support generic constructions over resources, such as "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 constructions.

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

---

[4]Our formalization exports nine rules for Hoare-style reasoning, each addressing an orthogonal linguistic feature.

[5]As apparent from the Morph rule, we explicitly bind the state s_w, in contrast to the classic presentation of separation logics where this state is implicit. This is merely a syntactic distinction.

[6]In addition to the examples from the paper, the sources include further benchmarks such as Treiber stack [Treiber 1986], flat combiner [Hendler et al. 2010], a concurrent allocator, a concurrent graph spanning tree algorithm [Sergey et al. 2015a], ticketed [Mellor-Crummey and Scott 1991] and readers-writers [Courtois et al. 1971] locks.

Readmap. The rest of the paper is organized as follows. Section 2 introduces resources and associated notions of ghost state and transitions, via the spin lock example. Section 3 develops the theory formally, including our specific notion of framing. Section 4 illustrates how to morph spin locks into exclusive locks. Section 5 introduces indexed morphism families and applies them to "forgetting" the ghost state of a resource. This models what is often referred to as *quiescence* [Aspnes et al. 1994; Derrick et al. 2011; Jagadeesan and Riely 2014; Nanevski et al. 2014; Sergey et al. 2016], most commonly used when installing one concurrent structure into the private state of another. Section 6 discusses related work and Section 7 concludes.

2 BACKGROUND AND OVERVIEW

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

2.1 Histories

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

Specifically, a program state s contains a *ghost* component that we project as $\tau_s s$, 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. 2015b].

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

2.2 Resources

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

The *state space* of Spin, denoted $\Sigma(\text{Spin})$, makes explicit the assumptions about the components: that the histories are disjoint (denoted $\tau_s s \perp \tau_o s$), that the entries in $\hat{\tau} s$ alternate between L and U, and that r isn't the null pointer.

The *erasure* $\ulcorner s \urcorner$ shows how the state s maps to a heap once the ghost histories are removed. The expression $r \mapsto \omega(\hat{\tau} s)$ denotes a heap with only the pointer r, storing the Boolean value $\omega(\hat{\tau} s)$. The latter computes the lock status out of the combined history $\hat{\tau} s$; it equals true if the last log in the combined history is a lock entry L, and false otherwise.

The *set of transitions* of Spin, denoted $\Delta(\text{Spin})$, contains lock_tr, unlock_tr, and the (elided) idle transition. The transition lock_tr adds a fresh L entry to $\tau_s s$ if $\neg\omega(\hat{\tau} s)$, i.e., if the lock is free in the pre-state. Similarly, unlock_tr adds a fresh U entry if $\omega(\hat{\tau} s)$, i.e., the lock is taken. The lock can by taken by "us" or by "others", as ω is computed from the combined history $\hat{\tau} s$. If the locking protocol insists that the thread that unlocks is the same thread that last locked, then the precondition of
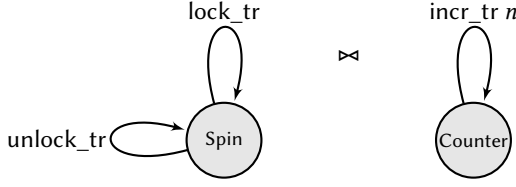
State space Σ (Spin): $s \in \Sigma$ (Spin) iff

    s contains fields $\tau_s\, s,\ \tau_o\, s \in$ Hist, and

    $\tau_s\, s \perp \tau_o\, s \wedge$ alternate $(\hat{\tau}\, s) \wedge r \neq$ null

Erasure: $\ulcorner s \urcorner \ \widehat{=}\ r \mapsto \omega\, (\hat{\tau}\, s)$

Transitions Δ(Spin):

lock_tr $s\ s'\ \widehat{=}\ \neg\omega\, (\hat{\tau}\, s) \wedge \tau_s\, s' = \tau_s\, s \bullet \text{fresh}\, (\hat{\tau}\, s) \mapsto \mathsf{L}$

unlock_tr $s\ s'\ \widehat{=}\ \omega\, (\hat{\tau}\, s) \wedge \tau_s\, s' = \tau_s\, s \bullet \text{fresh}\, (\hat{\tau}\, s) \mapsto \mathsf{U}$

State space Σ (Counter): $s \in \Sigma$ (Counter) iff

    s contains fields $\kappa_s\, s,\ \kappa_o\, s \in \mathbb{N}$

    with no additional constraints

Erasure: $\ulcorner s \urcorner \ \widehat{=}\ $ empty heap

Transitions Δ (Counter):

incr_tr $n\ s\ s'\ \widehat{=}\ \kappa_s\, s' = \kappa_s\, s + n$

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

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

$$\text{last\_stamp}\, h \ \widehat{=}\ \max\, (\{0\} \cup \text{dom}\, h)$$

$$\text{fresh}\, h \ \widehat{=}\ 1 + \text{last\_stamp}\, h \qquad \omega\, h \ \widehat{=}\ (\text{last\_op}\, h = \mathsf{L})$$

$$\text{alternate}\, h \ \widehat{=}\ (h = 1 \mapsto \mathsf{L} \bullet 2 \mapsto \mathsf{U} \bullet 3 \mapsto \mathsf{L} \bullet \cdots \bullet \text{last\_stamp}\, h \mapsto \text{last\_op}\, h)$$

Fig. 3. Spin and Counter resources (with id_tr elided).

unlock_tr should be changed to $\omega\, (\tau_s\, s)$. We don't want to impose such behavior at this stage, but show how to achieve it *a posteriori*, together with additional functionality, in Section 4.

2.3 Method Specifications

We now give the following pidgin code for lock and unlock, intended to further the intuition about transitions. The actual implementation of the methods will be shown in Section 3, once we have formally introduced our system.

$$\text{lock} \ \widehat{=}\ \text{do}\ \langle x \leftarrow \text{CAS}(r, \text{false}, \text{true});\ \boxed{\text{if } x \text{ then } \tau_s\, s := \tau_s\, s \bullet \text{fresh}\, (\hat{\tau}\, s) \mapsto \mathsf{L}}\, \rangle; \text{while } \neg x$$

$$\text{unlock} \ \widehat{=}\ \langle\ \boxed{x \leftarrow !r;\ r := \text{false};}\ \boxed{\text{if } x \text{ then } \tau_s\, s := \tau_s\, s \bullet \text{fresh}\, (\hat{\tau}\, s) \mapsto \mathsf{U}}\ \rangle$$

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

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

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

The precondition of lock starts with self history $\tau_s\, s$ equal to h[7], which is increased in the postcondition to log a locking event at time t. The conjunct $k < t$ in the postcondition claims that t is fresh, because it's larger than any k generated prior to the call (as $k \leq$ 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 $k \leq$ 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 (conjunct $\hat{\tau}\, s\, t = $ U). The conjunct $k \leq t$ captures that another thread may have freed r after the invocation of unlock ($k < t$), or that we invoked unlock with r already freed ($k = t$).

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

2.4 Morphisms

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

First, we "tensor" the resources Spin and Counter, as graphically indicated[8] in Figure 3; that is, we create a new resource SC whose state is a pair of Spin and Counter states, and transitions are lock_tr ⋈ incr_tr n and unlock_tr ⋈ id_tr. Operator ⋈ (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, for each $n \in \mathbb{N}$, we define the morphism $f_n : $ Spin $\rightarrow$ SC as follows:

$$\begin{aligned}
(f_n)_\Sigma\, s &\;\widehat{=}\; s\backslash 1 \\
(f_n)_\Delta\, s\, \text{lock\_tr} &\;\widehat{=}\; \text{lock\_tr} \bowtie \text{incr\_tr}\, n \\
(f_n)_\Delta\, s\, \text{unlock\_tr} &\;\widehat{=}\; \text{unlock\_tr} \bowtie \text{id\_tr}
\end{aligned}$$

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

Now, our desired program is

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

---

[7] 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.
[8] We elide the definition of tensoring, as it isn't required to follow the presentation. It can be found in the online appendices [Nanevski et al. 2019b, Appendix A].

1. $\{\kappa_s \, s = n\}$
2. $\{\kappa_s \, s = n \wedge \tau_s \, s = h\}$
3. morph f_1 lock; // $I_1 \, s \; \widehat{=} \; \kappa_s \, s = n + \sharp_L \, (\tau_s \, s) - \sharp_L \, h$
4. $\{\kappa_s \, s = n + \sharp_L \, (\tau_s \, s) - \sharp_L \, h \wedge \tau_s \, s = h \bullet t \Mapsto L\}$
5. $\{\kappa_s \, s = n + 1 \wedge \tau_s \, s = h'\}$
6. morph f_{42} unlock; // $I_{42} \, s \; \widehat{=} \; \kappa_s \, s = n + 1$
7. $\{\kappa_s \, s = n + 1 \wedge (\tau_s \, s = h' \vee \tau_s \, s = h' \bullet t' \Mapsto U)\}$
8. $\{\kappa_s \, s = n + 1 \wedge \tau_s \, s = h''\}$
9. morph f_2 lock // $I_2 \, s \; \widehat{=} \; \kappa_s \, s = n + 1 + 2(\sharp_L \, (\tau_s \, s) - \sharp_L \, h'')$
10. $\{\kappa_s \, s = n + 1 + 2(\sharp_L \, (\tau_s \, s) - \sharp_L \, h'') \wedge \tau_s \, s = h'' \bullet t'' \Mapsto L\}$
11. $\{\kappa_s \, s = n + 3\}$

Fig. 4. Using the MORPH rule to show that $\kappa_s \, s$ increments by 3. $\sharp_L \, (-)$ is the number of L-entries in a history.

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

2.5 Inference

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

$$\text{morph } f_1 \text{ lock;}$$
$$\text{morph } f_{42} \text{ unlock;}$$
$$\text{morph } f_2 \text{ lock}$$

which, in addition to locking and unlocking, increments $\kappa_s \, s$ by 1 in the first line, and by 2 in the third line.[9] The second line morphs unlock *vacuously*, as unlocking leaves $\kappa_s \, s$ unchanged. Nevertheless, *some* morphing of unlock is necessary, to bring the commands under the same resource type.

The proof outline in Figure 4 shows that $\kappa_s \, s$ increments by 3, and we discuss its main points next. In the outline, $\sharp_L$ is a function on history that computes the number of L entries in the history. The outline starts with the precondition $\kappa_s \, s = n$, where n snapshots "our" current count. Line 2 uses h to snapshot "our" history. Line 3 applies morph to lock, and correspondingly, the MORPH rule in the proof. At this point, we choose the simulation I_1 as indicated in line 3, to state that the counter κ_s increments n by the number of *fresh* L-entries in the history. Intuitively, I_1 is an f_1-simulation because it is preserved under incrementing $\kappa_s \, s$ by 1 while simultaneously adding an L-entry to $\tau_s \, s$ (Figure 5). It's easy to see that I_1 holds in line 2, thus by MORPH, it holds in line 4 as well. But, in line 4, by postcondition of lock, the history $\tau_s \, s$ has one more locking entry. Thus, $\kappa_s \, s$ is increased by 1 (line 5). The remainder of the outline proceeds similarly.

We close the discussion with the observation that the property of being a simulation (i.e., making diagrams in Figures 2 and 5 commute) relies only on the resource in the program's type, and the morphism in question, not on the program's code, as required for compositional reasoning. In this respect, the simulations are different from loop invariants, which are properties of programs. The MORPH rule ties the simulations to the morphed program by conjoining them with the program's pre- and the postcondition. Specifically above, I_1 enables computing the end-value of κ_s from the end-value of τ_s, and τ_s is given by the spec of lock.

---

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

$$
\begin{array}{ccc}
s'\backslash 1 \xleftarrow{\ (f_1)_\Sigma\ } I_1\, s' & \qquad & \kappa_s\, s' = n + \sharp_{\mathsf L}\,(\tau_s\, s') - \sharp_{\mathsf L}\, h \\
\Big\uparrow{\scriptstyle \text{lock\_tr}} \qquad \Big\uparrow {\scriptstyle \begin{array}{c}\text{lock\_tr}\\ \bowtie\\ \text{incr\_tr } 1\end{array}} & & \begin{array}{c}\tau_s\, s' = \tau_s\, s \bullet \mathrm{fresh}\,(\hat\tau\, s) \Mapsto \mathsf L\\ \kappa_s\, s' = \kappa_s\, s + 1\end{array} \\
s\backslash 1 \xleftarrow{\ (f_1)_\Sigma\ } I_1\, s & & \kappa_s\, s = n + \sharp_{\mathsf L}\,(\tau_s\, s) - \sharp_{\mathsf L}\, h
\end{array}
$$
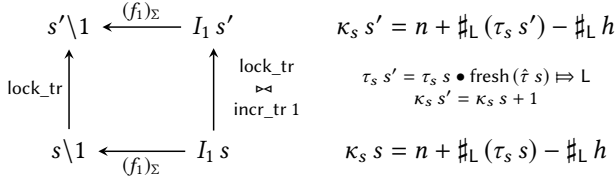
Fig. 5. Diagram showing that $I_1\, s \mathrel{\widehat{=}} \kappa_s\, s = n + \sharp_{\mathsf L}\,(\tau_s\, s) - \sharp_{\mathsf L}\, h$ is an f_1-simulation (case of lock_tr transition). The diagram specializes Figure 2 to f_1, I_1 and lock_tr.

3 DEFINITIONS OF THE FORMAL STRUCTURES

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

3.1 States

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

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

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

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

$$x \Mapsto 3 \bullet y \Mapsto \mathrm{false}$$

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

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

Imagine three threads θ_1, θ_2 and θ_3 running concurrently. Their respective states must have the forms $s_1 = (a_1, a_j, a_2 \bullet a_3)$, $s_2 = (a_2, a_j, a_3 \bullet a_1)$ and $s_3 = (a_3, a_j, a_1 \bullet a_2)$. Indeed, any two of the threads combined are the environment for the third thread. Thus, the PCM join of the *self*'s of any

---

[10] We silently already used this notation to define the erasure function for Spin in Figure 3.
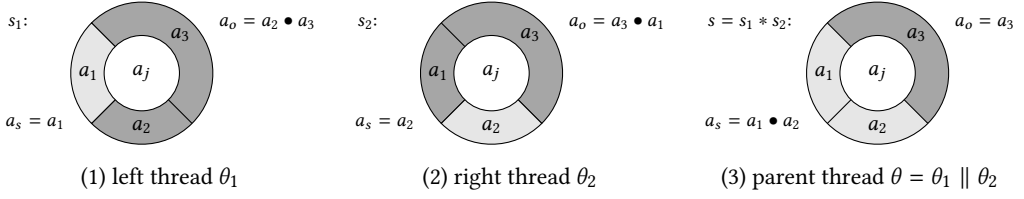
Fig. 6. Values of *self* component a_s (light shade) and *other* component a_o (dark shade) in the states of parallel threads and their parent. The inner white circle represents the *joint* component, and is equal for all threads.

two threads must equal the *other* of the third thread. Figures 6(1) and 6(2) illustrate this property for threads θ_1 and θ_2, with θ_3 being their implicit environment.

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, and illustrate it in Figure 6(3).

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

Definition 3.1. Let $p \in M$ 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 $s \rhd p = (a_s, a_j, p \bullet a_o)$.

Definition 3.2. A predicate P is **global**, if $P(s \rhd p) \leftrightarrow P(s \lhd p)$ for every p and state s 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.

Examples of global predicates from Section 2 are $P\, s = \tau_s\, s \perp \tau_o\, s$, and $Q\, s = \mathrm{alternate}\,(\hat{\tau}\, s)$ used in Figure 3 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 shuffling timestamps between τ_s and τ_o, as this doesn't alter the total. In fact, $\hat{\tau}$ itself is a global function, so we proceed to refer to $\hat{\tau}$ as the global history.

Definition 3.3. Let X be a global predicate. A predicate P is **global under** X, if $P(s \rhd p) \leftrightarrow P(s \lhd p)$ for every p and state s such that $a_s\, s \perp p \perp a_o\, s$ *and* $X(s \lhd p)$. Similarly for functions.

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

$$\mathrm{lock} : [k].\ \{\lambda s.\ \tau_s\, s = \emptyset \wedge k \leq \mathrm{last\_stamp}\,(\tau_o\, s)\}$$
$$\{\lambda s.\ \exists t.\ \tau_s\, s = t \mapsto \mathsf{L} \wedge k < t\}@\mathrm{Spin}$$

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

$$\mathrm{lock} : [h, k].\ \{\lambda s.\ \tau_s\, s = h \wedge k \leq \mathrm{last\_stamp}\,(\hat{\tau}\, s)\}$$
$$\{\lambda s.\ \exists t.\ \tau_s\, s = h \bullet t \mapsto \mathsf{L} \wedge k < t\}@\mathrm{Spin}$$

As expected in separation logic, framing increased the starting $\tau_s\, s$ from $\emptyset$ to h, which is the key distinction between small and large specs. But this isn't all it did; it also deducted h from $\tau_o\, s$. Indeed, had $\tau_o\, s$ been unchanged (as might also be expected in separation logic), then both specs would contain the same conjunct $k \leq \mathsf{last\_stamp}\,(\tau_o\, s)$. But the large spec contains $k \leq \mathsf{last\_stamp}\,(\hat{\tau}\, s) = \mathsf{last\_stamp}\,(h \bullet \tau_o\, s)$, where h is joined to $\tau_o\, s$ to compensate for the deduction.

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

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

3.2 Resources

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

Definition 3.4. A ***state space*** is a pair $\Sigma = (P, \ulcorner - \urcorner)$, where P is a global predicate and $\ulcorner - \urcorner$ is a partial function into heaps, global under P, 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 $P\, s$.

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

Definition 3.5. A ***transition*** t over state space 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 (2) captures that a transition can't change the *other*-component, as it's private to other threads. However, a transition can read this component, cf. how lock_tr in Figure 3 uses $\tau_o\, s$ as part of $\hat{\tau}\, s$ to compute a fresh timestamp.

Transition locality (3) essentially says that transitions can be framed. To see how, let 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 $s \lhd p$, by Figure 6. 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 $s \lhd p$. In other words, the transition of a child can be seen as a transition of the parent, but with *self* enlarged by p, and *other* suitably reduced by p. This is precisely the view of framing described in Section 3.1.5. Hence, transition locality is the base case of, and gives rise to, framing on programs, as a program's execution is a sequence of transitions.

Definition 3.6. A 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 3, which is footprint preserving, as it doesn't allocate or deallocate any pointers. Were it also to allocate, we will have a problem when combining Spin and Counter, as

we must impose that incr_tr won't allocate the pointer r, already taken by Spin. For simplicity, we here present the theory with only footprint-preserving transitions, but have added non-preserving (aka. external) transitions as well [Nanevski et al. 2019a]. External transitions encode transfer of data in and out of a resource [de Alfaro and Henzinger 2001], of which allocation and deallocation are an instance. When a resource requires allocation or deallocation, it can be tensored with an allocator resource to exchange pointers through ownership transfer [Filipović et al. 2010b; Nanevski et al. 2014] via external transitions. We elide further discussion, but refer to the Coq files for the implementation of an allocator resource and example programs that use it.

Definition 3.7. A **resource** is a tuple $V = (M, T, \Sigma, \Delta)$, where Σ is a space of (M, T)-states, 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 3.8. 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 $a (s \triangleright p) = (t, v)$ then $a (s \triangleleft p) = (t, v)$; that is, if a is performed by a child thread, it behaves the same when viewed by the parent.

The **effect** of a is the partial function $[a] : \Sigma(V) \rightharpoonup \Sigma(V) \times A$ 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:

$$\text{trylock\_act } s \ \widehat{=} \ \begin{cases} (\text{lock\_tr}, \text{true}) & \text{if } \neg\omega(\hat{\tau} s) \\ (\text{id\_tr}, \text{false}) & \text{otherwise} \end{cases} \tag{1}$$

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$(r, \text{false}, \text{true})$, as expected.[11] Similarly,

$$\text{unlock\_act } s \ \widehat{=} \ \begin{cases} (\text{unlock\_tr}, ()) & \text{if } \omega(\hat{\tau} s) \\ (\text{id\_tr}, ()) & \text{otherwise} \end{cases} \tag{2}$$

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

3.3 Morphisms

Definition 3.9. A **resource morphism** $f : V \to W$ consists of two partial functions $f_\Sigma : \Sigma(W) \rightharpoonup \Sigma(V)$ (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 $\phi : M(W) \rightharpoonup M(V)$ such that if $f_\Sigma (s_w \triangleright p) = s_v$, then there exists s'_v such that $s_v = s'_v \triangleright \phi(p)$, and $f_\Sigma (s_w \triangleleft p) = s'_v \triangleleft \phi(p)$.
(2) *(locality of f_Δ)* if $f_\Delta (s_w \triangleright p)(t_v) = t_w$, then $f_\Delta (s_w \triangleleft p)(t_v) = t_w$.
(3) *(other-fixity)* if $a_o (s_w) = a_o (s'_w)$ and $f_\Sigma (s_w)$, $f_\Sigma (s'_w)$ exist, then $a_o (f_\Sigma (s_w)) = a_o (f_\Sigma (s'_w))$.

---

[11]All the actions we use in this paper and in the Coq code erase to some RMW command. However, we proved this only by hand, as our formalism and the Coq implementation don't currently issue proof obligations to check this. In general, we currently treat code and ghost code equally, and, as customary in type theory, equally to proofs. Differentiating between these *formally* is an orthogonal issue that we plan to address in the future by making a type distinction between them, such as in the work on proof irrelevance in type theory [Barras and Bernardo 2008; Gilbert et al. 2019; Pfenning 2001].

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

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

Example. Properties (1)-(3) are all satisfied by the morphisms $f_n : \text{Spin} \to \text{SC}$ from Section 2. Indeed, $M(\text{SC}) = M(\text{Spin}) \times M(\text{Counter}) = \text{Hist} \times \mathbb{N}$. Thus, a frame in 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).

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

Definition 3.10. The ***identity*** morphism $\text{id} : V \to V$ is defined by $\text{id}_\Sigma\ s = s$ and $\text{id}_\Delta\ s\ t = t$. The ***composition*** of morphisms $f : U \to V$ and $g : V \to W$ is the morphism $g \circ f : U \to W$ defined by:

$$(g \circ f)_\Sigma\ s_w \quad \widehat{=}\quad f_\Sigma(g_\Sigma\ s_w)$$
$$(g \circ f)_\Delta\ s_w\ t_u \quad \widehat{=}\quad g_\Delta\ s_w\ (f_\Delta\ (g_\Sigma\ s_w)\ t_u)$$

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 typechecks against the MORPH rule doesn't get stuck.

Definition 3.11. Given a morphism $f : V \to W$, an f-***simulation*** is a predicate I on W-states such that:

(1) if $I\ s_w$, and $s_v = f_\Sigma(s_w)$ exists, and $t_v\ s_v\ s'_v$, then there exist $t_w = f_\Delta\ s_w\ t_v$ and s'_w such that $I\ s'_w$ and $s'_v = f_\Sigma(s'_w)$, and $t_w\ s_w\ s'_w$.

(2) if $I\ s_w$, and $s_v = f_\Sigma(s_w)$ exists, and $s_w \xrightarrow[W]{}^* s'_w$, then $I\ s'_w$, and $s'_v = f_\Sigma(s'_w)$ exists, and $s_v \xrightarrow[V]{}^* s'_v$. 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 ***transposition*** $s^\top = (a_o\ s, a_j\ s, a_s\ s)$ swaps the subjective components of s, to obtain the view of *other* threads. The relation $\xrightarrow[W]{}^*$ is the reflexive-transitive closure of $\xrightarrow[W]{}$, allowing for an arbitrary number of steps.
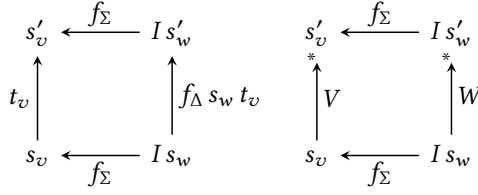
Fig. 7. Commutative diagrams for the properties (1) and (2) of Definition 3.11 for I to be an f-simulation.

Property (1) says that W simulates V on states satisfying I. Property (2) states the simulation in the opposite direction, i.e., of W by V, but allowing many *other*-steps to match many *other*-steps. Notice that *other*-stepping transitions over *transposed* states; that is, it changes the *other*, but, by Definition 3.5(2), preserves the *self* of the states. Intuitively, (2) ensures that interference in W may be viewed as interference in V, so that stable Hoare triples in V can be transformed into stable Hoare triples in W, which is required for the soundness of the Morph rule. Property (1) has already been shown in Figure 2; we repeat it in Figure 7, together with a diagram for property (2).

3.5 Inference Rules

We present the system using the Calculus of Inductive Constructions (CIC) as an environment logic, hence as a shallow embedding in Coq. We inherit from CIC the useful concepts of higher-order functions and substitution principles, and only present the notions specific to Hoare logic[12].

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

The key concept in the inference rules is the predicate transformer vrf $e\ Q$, which takes a program $e : ST\ V\ A$, and a postcondition Q, and returns the set of V-states from which e is safe to run[13] and produces a result v and ending state s' such that $Q\ v\ s'$. Hoare types are then defined in terms of vrf, as follows.

$$[\Gamma].\ \{P\}\ A\ \{Q\}@V = \{e : ST\ V\ A \mid \forall\Gamma.\ \forall s \in \Sigma\ (V).\ P\ s \rightarrow \text{vrf } e\ Q\ s\} \qquad (3)$$

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

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

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

In reasoning about programs, we keep the transformer vrf abstract, and only rely on the following minimal set of rules. These, together with the above definition of Hoare types and typing for fix,

---

[12]The extended version of the paper [Nanevski et al. 2019b, Appendix D] defines the denotational semantics, in CIC, for these notions, and states a theorem, proved in Coq, that the inference rules are sound wrt. the denotational semantics.
[13]Thus ensuring fault avoidance.

are the only Hoare-related rules of the system. In the rules we assume that $e : ST\ V\ A$, $e_i : ST\ V\ A_i$, a is a V-action, $f : V \rightarrow W$ is a morphism, I is an f-simulation, $s \in \Sigma(V)$, and $s_w \in \Sigma(W)$.

$$
\begin{aligned}
&\text{vrf\_post} &&: (\forall v\ s.\ J\ s \rightarrow Q_1\ v\ s \rightarrow Q_2\ v\ s) \rightarrow J\ s \rightarrow \text{vrf}\ e\ Q_1\ s \rightarrow \text{vrf}\ e\ Q_2\ s \\
&\text{vrf\_ret} &&: (Q\ v)^\bullet\ s \rightarrow \text{vrf}\ (\text{ret}\ v)\ Q\ s \\
&\text{vrf\_bnd} &&: \text{vrf}\ e_1\ (\lambda x.\ \text{vrf}\ (e_2\ x)\ Q)\ s \rightarrow \text{vrf}\ (x \leftarrow e_1; (e_2\ x))\ Q\ s \\
&\text{vrf\_par} &&: ((\text{vrf}\ e_1\ Q_1) * (\text{vrf}\ e_2\ Q_2))\ s \rightarrow \text{vrf}\ (e_1 \parallel e_2)\ (\lambda v{:}A_1 \times A_2.\ (Q_1\ v.1) * (Q_2\ v.2))\ s \\
& && \quad \text{where}\ (P * Q)\ s \;\widehat{=}\; \exists s_1\ s_2.\ s = s_1 * s_2 \wedge P\ s_1 \wedge Q\ s_2 \\
&\text{vrf\_frame} &&: ((\text{vrf}\ e\ Q_1) * Q_2^\bullet)\ s \rightarrow \text{vrf}\ e\ (\lambda v.\ (Q_1\ v) * Q_2)\ s \\
&\text{vrf\_act} &&: (\lambda s'.\ \exists s''\ v.\ [a]\ s' = (s'', v) \wedge (Q\ v)^\bullet\ s'')^\bullet\ s \rightarrow \text{vrf}\ \langle a \rangle\ Q\ s \\
&\text{vrf\_morph} &&: \hat{f}(\text{vrf}\ e\ Q)\ s_w \rightarrow I\ s_w \rightarrow \text{vrf}\ (\text{morph}\ f\ e)\ (\lambda v\ s'_w.\ \hat{f}(Q\ v)\ s'_w \wedge I\ s'_w)\ s_w \\
& && \quad \text{where}\ \hat{f}R\ s_w \;\widehat{=}\; \exists s_v.\ s_v = f_\Sigma\ s_w \wedge R\ s_v
\end{aligned}
$$

In English:

- The vrf_post rule weakens the postcondition, similar to the well-known rule of Consequence in Hoare logic. The rule allows assuming a property J when establishing a postcondition Q_2 out of Q_1. Here J is an **invariant**, i.e., a property preserved by the transitions of V; an id-simulation. Thus, invariants can be elided from program specs, and invoked by vrf_post when needed.

- The vrf_ret rule applies to an idle program returning v. When we want an idle program that returns no value, we simply take v to be of unit type. The rule explicitly **stabilizes** the postcondition Q to allow for the state s to be changed by interference of other threads in between the invocation of the idle program and its termination. Here, stabilization of a predicate Q is $Q^\bullet(s) \;\widehat{=}\; \forall s'.\ s \xrightarrow[V]{*} s' \rightarrow Q(s')$. The predicate 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. In order to show that the sequential composition $x \leftarrow e_1; (e_2\ x)$ has a postcondition Q, it suffices to show that e_1 has a postcondition $\lambda x.\ \text{vrf}\ (e_2\ x)\ Q$. In other words, e_1 terminates with a value x and in a state satisfying vrf $(e_2\ x)\ Q$, so that running $e_2\ x$ in that state yields Q.

- The vrf_par and vrf_frame rules are predicate transformer variants of the rules for parallel composition and framing from separation logic. The separating conjunction $P * Q$ is defined as customary in separation logic, except that we use the subjective splitting of state, as explained in Section 3.1.3 and Figure 6. The vrf_frame rule can be seen as an instance of vrf_par, where e_2 is taken to be the idle programs returning no value. Thus, Q_2 is explicitly stabilized in vrf_frame, to match the precondition of the vrf_ret rule for idle programs.

- The vrf_act rule says that Q holds after executing action a in state s, if s steps to s' by interfering threads, and then $[a]\ s'$ returns the pair (s'', v) of output state and value. The latter satisfy the stabilization of Q, to allow for interference on s'' after the termination of a.

- The vrf_morph rule is a straightforward casting of the Morph rule from Section 1 into a predicate transformer style.

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

$$
\text{vrf\_cond} : (\text{if}\ b\ \text{then}\ \text{vrf}\ e_1\ Q\ s\ \text{else}\ \text{vrf}\ e_2\ Q\ s) \rightarrow \text{vrf}\ (\text{if}\ b\ \text{then}\ e_1\ \text{else}\ e_2)\ Q\ s
$$

All the other customary rules of Hoare logic also become derivable. For example, if $e : \{P\}\ \{Q\}$ and $\forall s \in \Sigma(V).\ P'\ s \rightarrow P\ s$, then also $e : \{P'\}\ \{Q\}$. Similarly, if e depends on a logical variable $x : A$

$tyLck \mathrel{\widehat{=}} [h, k]. \{\lambda s. \tau_s\, s = h \wedge k \leq \mathsf{last\_stamp}\,(\hat{\tau}\, s)\}\{\lambda s. \exists t. \tau_s\, s = h \bullet t \mapsto \mathsf{L} \wedge k < t\}@\mathsf{Spin}$

1. $\mathsf{fix}\ (\lambda\, loop : \mathsf{unit} \rightarrow tyLck.\ \lambda\_ : \mathsf{unit}.$
2. $\quad \{\tau_s\, s = h \wedge k \leq \mathsf{last\_stamp}\,(\hat{\tau}\, s)\}$
3. $\quad \{\exists s'\, b.\, [\mathsf{trylock\_act}]\, s = (s', b)\ \wedge$
   $\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ \exists t.\, \tau_s\, s' = h \bullet t \mapsto \mathsf{L} \wedge k < t\ \mathsf{else}\ \tau_s\, s' = h \wedge k \leq \mathsf{last\_stamp}\,(\hat{\tau}\, s')\}$
4. $\quad b \leftarrow \langle \mathsf{trylock\_act}\rangle;$
5. $\quad \{\mathsf{if}\ b\ \mathsf{then}\ \exists t.\, \tau_s\, s = h \bullet t \mapsto \mathsf{L} \wedge k < t\ \mathsf{else}\ \tau_s\, s = h \wedge k \leq \mathsf{last\_stamp}\,(\hat{\tau}\, s)\}$
6. $\quad \mathsf{if}\ b\ \mathsf{then}\ \{\exists t.\, \tau_s\, s = h \bullet t \mapsto \mathsf{L} \wedge k < t\}\ \mathsf{ret}\ ()\ \{\exists t.\, \tau_s\, s = h \bullet t \mapsto \mathsf{L} \wedge k < t\}$
7. $\quad \mathsf{else}\ \{\tau_s\, s = h \wedge k \leq \mathsf{last\_stamp}\,(\hat{\tau}\, s)\}\ loop\,()\ \{\exists t.\, \tau_s\, s = h \bullet t \mapsto \mathsf{L} \wedge k < t\}$
8. $\quad \{\exists t.\, \tau_s\, s = h \bullet t \mapsto \mathsf{L} \wedge k < t\})\ ()$

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

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

3.6 Revisiting Spinlocks

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

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

4 EXCLUSIVE LOCKING VIA MORPHING AND THE NEED FOR PERMISSIONS

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

---

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

[15]In the case of failure, we could also derive that the lock was taken at the moment trylock_act was attempted, i.e. $\exists t.\, \tau_s\, s' = h \wedge \hat{\tau}\, s'\, t = \mathsf{L} \wedge k \leq t \leq \mathsf{last\_stamp}\,(\hat{\tau}\, s')$. However, the rest of the proof doesn't require the additional detail.
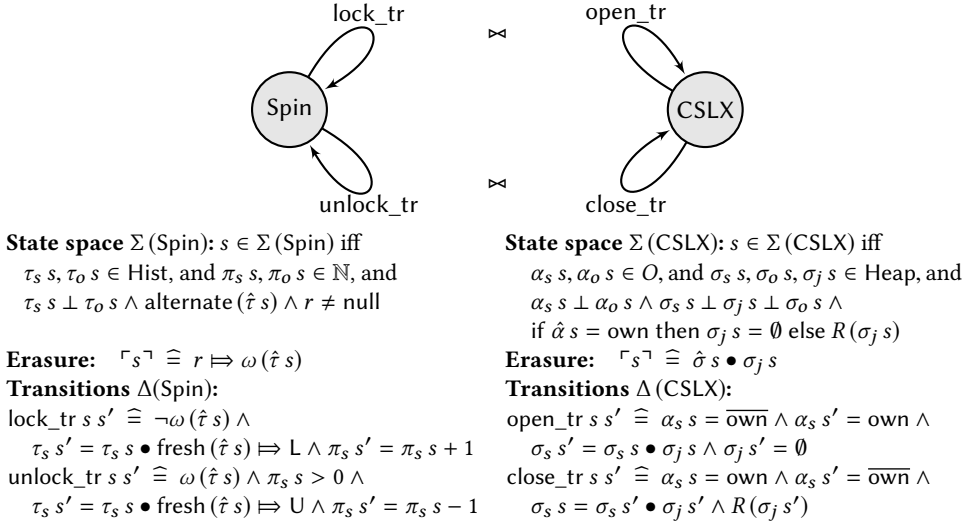
State space Σ (Spin): $s \in \Sigma$ (Spin) iff
   $\tau_s\, s, \tau_o\, s \in$ Hist, and $\pi_s\, s, \pi_o\, s \in \mathbb{N}$, and
   $\tau_s\, s \perp \tau_o\, s \wedge$ alternate $(\hat{\tau}\, s) \wedge r \neq$ null

Erasure: $\ulcorner s \urcorner \mathrel{\widehat{=}} r \mapsto \omega\,(\hat{\tau}\, s)$
Transitions Δ(Spin):
lock_tr $s\, s' \mathrel{\widehat{=}} \neg\omega\,(\hat{\tau}\, s) \wedge$
   $\tau_s\, s' = \tau_s\, s \bullet$ fresh $(\hat{\tau}\, s) \Mapsto \mathsf{L} \wedge \pi_s\, s' = \pi_s\, s + 1$
unlock_tr $s\, s' \mathrel{\widehat{=}} \omega\,(\hat{\tau}\, s) \wedge \pi_s\, s > 0 \wedge$
   $\tau_s\, s' = \tau_s\, s \bullet$ fresh $(\hat{\tau}\, s) \Mapsto \mathsf{U} \wedge \pi_s\, s' = \pi_s\, s - 1$

State space Σ (CSLX): $s \in \Sigma$ (CSLX) iff
   $\alpha_s\, s, \alpha_o\, s \in O$, and $\sigma_s\, s, \sigma_o\, s, \sigma_j\, s \in$ Heap, and
   $\alpha_s\, s \perp \alpha_o\, s \wedge \sigma_s\, s \perp \sigma_j\, s \perp \sigma_o\, s \wedge$
   if $\hat{\alpha}\, s =$ own then $\sigma_j\, s = \emptyset$ else $R\,(\sigma_j\, s)$

Erasure: $\ulcorner s \urcorner \mathrel{\widehat{=}} \hat{\sigma}\, s \bullet \sigma_j\, s$
Transitions Δ(CSLX):
open_tr $s\, s' \mathrel{\widehat{=}} \alpha_s\, s = \overline{\mathsf{own}} \wedge \alpha_s\, s' =$ own $\wedge$
   $\sigma_s\, s' = \sigma_s\, s \bullet \sigma_j\, s \wedge \sigma_j\, s' = \emptyset$
close_tr $s\, s' \mathrel{\widehat{=}} \alpha_s\, s =$ own $\wedge \alpha_s\, s' = \overline{\mathsf{own}} \wedge$
   $\sigma_s\, s = \sigma_s\, s' \bullet \sigma_j\, s' \wedge R\,(\sigma_j\, s')$

Fig. 9. 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).

locks from Section 2. An exclusive lock protects a shared heap, satisfying a user-supplied predicate R (*aka.* resource invariant). Upon successful locking, the shared heap is transferred to the private ownership of the locking thread, where it can be modified at will, potentially violating R. Before unlocking, the owning thread must re-establish R in its private heap, after which, the part of the heap satisfying R is moved back to the shared status. The idea is captured by the following methods and specs, which we name exlock and exunlock to differentiate from lock and unlock in Section 2.

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

Here $\mu_s\, s$ is a ghost 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 own $\bullet$ own undefined, to capture that the locking is exclusive, i.e., the lock can't be owned by a thread and its environment simultaneously. Our goal in this section is to *derive* exlock and exunlock using morphing and simulations to "attach" to lock and unlock the functionality of transferring the protected heap between shared and private state.

The idea for doing so is pictorially shown in Figure 9, where the CSLX resource contains the state components and transitions describing the functionality needed for heap transfers. In particular, $\alpha_s\, s \in O$ keeps track of whether we own the lock or not, 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 $\to$ CSL′ defined similarly to the morphisms in Section 2:

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

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 functions out of the state components of CSL.

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

The analysis implies that we should have defined Spin in a more general way, as shown in Figure 9. In particular, Spin should contain the integer components π_s/π_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*.[16],[17]

A thread may have more than one permission to unlock, which it can distribute among its children upon forking, who can then race to unlock. The addition of the new fields leads to the following minimal modification of the specs from Section 2, to indicate that lock enables unlock_tr, and a successful unlock consumes one permission. Note that the specs don't assume that having a permission to unlock implies that it was "us" who last locked, or even that the lock is taken. We will impose such locking-protocol specific properties on CSL, but there is no need for them in Spin.[18]

$$\text{lock}' : [h, k].\ \{\lambda s.\ \tau_s\ s = h \land k \leq \text{last\_stamp}\,(\hat{\tau}\ s) \land \pi_s\ s = 0\}$$
$$\{\lambda s.\ \exists t.\ \tau_s\ s = h \bullet t \mapsto \mathsf{L} \land k < t \land \pi_s\ s = 1\}@\text{Spin}$$
$$\text{unlock}' : [h, k].\ \{\lambda s.\ \tau_s\ s = h \land k \leq \text{last\_stamp}\,(\hat{\tau}\ s) \land \pi_s\ s = 1\}$$
$$\{\lambda s.\ \exists t.\ \tau_s\ s = h \bullet t \mapsto \mathsf{U} \land k < t \land \pi_s\ s = 0 \ \lor$$
$$\tau_s\ s = h \land \hat{\tau}\ s\ t = \mathsf{U} \land k \leq t \land \pi_s\ s = 1\}@\text{Spin}$$

Let us now consider the combination CSL′ of Spin and CSLX as defined in Figure 9. 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

$$\omega\,(\tau_s\ s) = (\alpha_s\ s = \text{own}) \tag{4}$$
$$\omega\,(\tau_o\ s) = (\alpha_o\ s = \text{own})$$

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

---

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

[17]Similar concepts arise in other concurrency models as well. For example, a transition in a Petri net fires only if there are sufficient tokens—akin to permissions—in its input places. The tokens are consumed upon firing.

[18]Following Section 3.1.5, we make the specs small wrt. $\pi_s\ s$, for simplicity. By framing, lock′ can be invoked when $\pi_s\ s \geq 0$, in which case it increments $\pi_s\ s$ by 1.

require that only the thread that holds the lock has the permission to unlock:

$$\pi_s \, s = (\text{if } \alpha_s \, s = \text{own then 1 else 0}) \tag{5}$$
$$\pi_o \, s = (\text{if } \alpha_o \, s = \text{own then 1 else 0})$$

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

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

$$\tau_s \, s \perp_\omega \tau_o \, s \tag{6}$$

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

$$(\omega \, h \rightarrow \text{last\_stamp} \, k < \text{last\_stamp} \, h) \, \wedge$$
$$(\omega \, k \rightarrow \text{last\_stamp} \, h < \text{last\_stamp} \, k) \wedge h \perp k$$

to say that if h (resp. k) indicates that a thread holds the lock, then another thread couldn't have proceeded to add logs to its own history k (resp. h), and unlock itself.[19]

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

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

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

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 $\Sigma \, (\text{CSL})$ to $\Sigma \, (\text{CSL}')$.

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

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

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

---

[19]Requirement (6) restricts only the *last* timestamp in h and k, not all timestamps hereditarily. This suffices for our proof.

the derivation for exunlock = morph $(\iota \circ f)$ unlock', and refer to the Coq code [Nanevski et al. 2019a] for the derivation of exlock, which is similar.

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

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

5 QUIESCENCE AND INDEXED MORPHISM FAMILIES

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

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

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

$$\frac{e : \{P\}\,\{Q\}@V}{\text{morph } f\, x\, e : \{\lambda s_w.\, (f\, x)\hat{}\, P\, s_w \land I\, x\, s_w\}\, \{\lambda s_w.\, \exists y.\, (f\, y)\hat{}\, Q\, s_w \land I\, y\, s_w\}@W} \; \text{MorphX}$$

$$\text{where } (f\, x)\hat{}\, R\, s_w \;\widehat{=}\; \exists s_v.\, s_v = f_\Sigma\, x\, s_w \land R\, s_v$$

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

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

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 joint heap σ_j stores the stack's physical layout, and $\hat{\alpha}\, s$ is the abstract

Stack:

$\text{dom}\,(\hat{\tau}\,s) = \{1, \ldots, \text{last\_stamp}\,(\hat{\tau}\,s)\}$

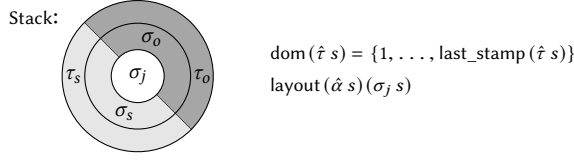$\text{layout}\,(\hat{\alpha}\,s)\,(\sigma_j\,s)$

Fig. 10. Representation of the state components of the Stack resource. The *self* components (σ_s for the heap, and τ_s for the history) are in light shade, the *other* components are in dark; the *joint* component (σ_j for the heap storing the stack's physical layout) is white. The abbreviation $\hat{\alpha}\,s$ is the abstract value of the stack (computed out of $\hat{\tau}\,s$).

contents of the stack as a mathematical sequence (computed out of $\hat{\tau}\,s$). Intuitively, push 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 $\hat{\alpha}$ to reflect the addition of the node. We also elide the full definition of Stack as it isn't essential here; it suffices to know that predicate layout describes how $\hat{\alpha}$ is laid out in σ_j (i.e., $\forall s \in \Sigma\,(\text{Stack}).\,\text{layout}\,(\hat{\alpha}\,s)\,(\sigma_j\,s)$), and that the global history $\hat{\tau}\,s$ has no timestamp gaps (i.e., $\forall s \in \Sigma\,(\text{Stack}).\,\text{dom}\,(\hat{\tau}\,s) = \{1, \ldots, \text{last\_stamp}\,(\hat{\tau}\,s)\}$).[20]

Consider now the program $e = \text{push}(a) \parallel \text{push}(b)$ of the following type (also derived in Coq):

$e : \{\lambda s.\ \sigma_s\,s = \emptyset \wedge \tau_s\,s = \emptyset\}$
$\quad\{\lambda s.\ \sigma_s\,s = \emptyset \wedge \exists t_1\ vs_1\ t_2\ vs_2.\ \tau_s\,s = t_1 \mapsto (vs_1, a :: vs_1) \bullet t_2 \mapsto (vs_2, b :: vs_2)\}@\text{Stack}$

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 χ_s and χ_o, and no other components (Figure 11(1)), 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]. A program working over Priv can install an empty stack in χ_s and then invoke e over it. Because the stack is installed *privately*, no threads other than the two children of e can race on it. Could we exploit quiescence, and derive *just out of the specification* of e that the stack at the end stores either the list $[a, b]$, or $[b, a]$? This fact can be stated even without histories, using solely heaps, as follows:

$$\{\lambda s.\ \text{layout nil}\,(\sigma_s\,s)\}\{\lambda s.\ \text{layout}\,[a, b]\,(\chi_s\,s) \vee \text{layout}\,[b, a]\,(\chi_s\,s)\}@\text{Priv} \qquad (7)$$

The move from Stack to Priv thus essentially *forgets* the ghost state of histories, and the distinction in Stack between shared and private heaps. These components and distinctions are visible when in the scope of Stack, but hidden when in Priv. We would like to obtain the spec (7) by applying the Morph rule to the Stack spec of e, with a morphism $g : \text{Stack} \to \text{Priv}$ that forgets the histories. Unfortunately, such a morphism can't be constructed as-is. Were it to exist, then g_Σ, being contravariant, should map a state s_{Priv}, containing only heaps, to a state s_{Stack}, containing heaps *and histories*; thus g_Σ must "invent" the history component out of thin air.

---

[20]We mechanized this development in Coq for the Treiber variant of stacks, with some minor Treiber-specific modifications.
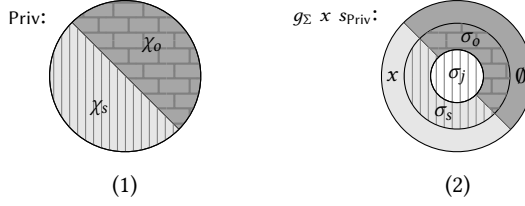
Fig. 11. The left figure shows the state components of Priv. The right figure shows how Stack from Figure 10 is related to Priv by g_Σ. Striped and bricked regions show the Priv's *self* heap $\chi_s = \sigma_s \bullet \sigma_j$ and *other* heap $\chi_o = \sigma_o$, respectively. Solid regions show the Stack's *self* history $\tau_s = x$ and *other* history $\tau_o = \emptyset$.

This is where families come in. We make $g : \text{Stack} \xrightarrow{\text{Hist}} \text{Priv}$ a family over $X = \text{Hist}$, thereby passing to g_Σ the history x that should be added to s_Priv to produce an s_Stack (Figure 11(2)).

$$g_\Sigma \ x \ s_\text{Priv} = s_\text{Stack} \ \widehat{=} \ \chi_s (s_\text{Priv}) = \sigma_s (s_\text{Stack}) \bullet \sigma_j (s_\text{Stack}) \wedge$$
$$\chi_o (s_\text{Priv}) = \sigma_o (s_\text{Stack}) \wedge$$
$$\tau_s (s_\text{Stack}) = x \wedge \tau_o (s_\text{Stack}) = \emptyset$$

The first conjunct directly states that Stack is installed in $\chi_s (s_\text{Priv})$ by making $\chi_s (s_\text{Priv})$ be the join of the heaps $\sigma_j (s_\text{Stack})$ and $\sigma_s (s_\text{Stack})$.[21] The second conjunct says that the heap $\chi_o (s_\text{Priv})$ of the interfering threads is propagated to $\sigma_o (s_\text{Stack})$. The third conjunct captures that the history component of s_Stack is set to the index x, as discussed immediately above. In the last conjunct, the $\tau_o (s_\text{Stack})$ history is declared $\emptyset$, thus formalizing quiescence. We elide the definition of g_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.

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

1. $\{\text{layout nil} (\chi_s (s_\text{Priv}))\}$
2. $\{\exists s_\text{Stack}. \ s_\text{Stack} = g_\Sigma \ \emptyset \ s_\text{Priv} \wedge \sigma_s (s_\text{Stack}) = \emptyset \wedge \tau_s (s_\text{Stack}) = \emptyset\}$
3. $\text{morph} \ g \ \emptyset \ e$
4. $\{\exists y \ s_\text{Stack}. \ s_\text{Stack} = g_\Sigma \ y \ s_\text{Priv} \wedge \sigma_s (s_\text{Stack}) = \emptyset \wedge$
   $\qquad \exists t_1 \ vs_1 \ t_2 \ vs_2. \ \tau_s (s_\text{Stack}) = t_1 \mapsto (vs_1, a :: vs_1) \bullet t_2 \mapsto (vs_2, b :: vs_2)\}$
5. $\{\text{layout} \ [a, b] (\chi_s (s_\text{Priv})) \vee \text{layout} \ [b, a] (\chi_s (s_\text{Priv}))\}$

Line 2 derives trivially from line 1, as the state s_Stack is uniquely determined; i.e., take $\sigma_j (s_\text{Stack})$ to be the heap storing the empty stack, and all the other components of s_Stack to be empty. We next derive the postcondition in line 5. From the properties of $\Sigma (\text{Stack})$, we know layout $(\hat{\alpha} (s_\text{Stack})) (\sigma_j (s_\text{Stack}))$. From $s_\text{Stack} = g_\Sigma \ y \ s_\text{Priv}$ and other conjuncts in line 4, we know $\chi_s (s_\text{Priv}) = \sigma_j (s_\text{Stack})$, and $\tau_s (s_\text{Stack}) = y$, and $\tau_o (s_\text{Stack}) = \emptyset$. Thus, it is also $\hat{\tau} (s_\text{Stack}) = y = t_1 \mapsto (vs_1, a :: vs_1) \bullet t_2 \mapsto (vs_2, b :: vs_2)$, and:

$$\text{layout} (\hat{\alpha} (s_\text{Stack})) (\chi_s (s_\text{Priv})) \tag{8}$$

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

---

[21]As we want to build s_Stack out of s_Priv, we have to identify the part of $\chi_s (s_\text{Priv})$ which we want to assign to $\sigma_j (s_\text{Stack})$. This part has to be uniquely determined, else g_Σ won't be a function. We ensure uniqueness by insisting that the predicate layout is precise – a property commonly required in separation logics.

6 RELATED WORK

Coalgebra morphisms and refinement mappings. Transition systems are mathematically representable as coalgebras [Jacobs 2016; Rutten 2000]; thus, resource morphisms are closely related to coalgebra morphisms, with differences arising from our application to concurrent separation logic and types. For example, for us, given $f : V \to W$, 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 preserve and reflect the V-transitions (i.e., if $x \to y$ is a V-transition, then $f(x) \to f(y)$ is a W-transition, and if $f(x) \to t$ is a W-transition, then there exists y such that $x \to y$ is a V-transition, and $f(y) = t$). These properties are similar in spirit to the two clauses of our Definition 3.11 of f-simulations.

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

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

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

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

threads, whereas linearizability is typically considered on programs with a fixed, though arbitrary, number of threads.

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

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

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

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

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

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

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

By relying on these foundational abstractions, we achieve some uniformity and simplicity of reasoning, which we expect to build on in the future. For example, one challenge to parametrization arises when the point at which to execute the ghost function can be determined only after the program has terminated. This is a common pattern when proving linearizability, as exhibited, say, by the queue of Herlihy and Wing [1990], but has been difficult to address by parametrization,

because it isn't clear at which point in the code to invoke the parametrizing ghost function. In our case, histories separate the termination of operations from their order in the linearization. The order becomes just another ghost component that can be constructed at run time, with cooperation of other threads [Delbianco et al. 2017]. A history-based spec can say that an operation finished executing, but that its exact place in the linearization is to be fully determined only later, by the action of other threads. We thus expect that history-based specs will support the described pattern of linearizability proofs.

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

Sergey et al. [2018] have designed a logic Disel for distributed systems, in which one can combine distributed protocols—represented as STSs—by means of *hooks*. A hook on a transition t prevents t from execution, unless the condition P associated with the hook is satisfied. In this sense, hooks implement an instance of our transition coupling, where one operand is fixed to the idle transition with a condition P, i.e. 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.

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

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

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

7 CONCLUSIONS

This paper develops novel notions of resource morphisms and associated simulations, as key mathematical concepts that underpin a separation logic for fine-grained concurrency. This is a natural development, as structures in mathematics are always associated with an appropriate notion of morphism, and simulations are the invariants that the morphisms preserve. Morphisms and simulations act on programs, and are integrated into separation logic via a single inference rule that propagates the simulation from the precondition to the postcondition of the morphed program.

Morphisms compose and can support different constructions and applications. One application is abstract atomicity, whereby a general spec, such as the one for lock in Section 2, is specialized to a specific ownership discipline, e.g., exclusive locking in Section 4. Other applications include the managing of scope of ghost state in quiescent environments, as illustrated in Section 5, and restricting the state space of a resource with additional state and PCM invariants, as used in Section 4.

REFERENCES

Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science (TCS)* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

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

James Aspnes, Maurice Herlihy, and Nir Shavit. 1994. Counting Networks. *J. ACM* 41, 5 (1994), 1020–1048. https://doi.org/10.1145/185675.185815

Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. 365–379. https://doi.org/10.1007/978-3-540-78499-9_26

Stefan Blom and Marieke Huisman. 2014. The VerCors tool for verification of concurrent programs. In *International Symposium on Formal Methods (FM) (LNCS)*, Vol. 8442. 127–131. https://doi.org/10.1007/978-3-319-06410-9_9

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. https://doi.org/10.1145/1040305.1040327

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. Proving linearizability using forward simulations. In *Computer Aided Verification (CAV)*. 542–563. https://doi.org/10.1007/978-3-319-63390-9_28

Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theoretical Computer Science (TCS)* 375, 1-3 (2007), 227–270. https://doi.org/10.1016/j.tcs.2006.12.034

Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2014. Parameterised linearisability. In *International Colloquium on Automata, Languages and Programming (ICALP)*. 98–109. https://doi.org/10.1007/978-3-662-43951-7_9

P. J. Courtois, F. Heymans, and D. L. Parnas. 1971. Concurrent control with "readers" and "writers". *Commun. ACM* 14, 10 (1971), 667–668. https://doi.org/10.1145/362759.362813

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. https://doi.org/10.1007/978-3-662-44202-9_9

Luca de Alfaro and Thomas A. Henzinger. 2001. Interface Automata. *ACM Software Engineering Notes (SEN)* 26, 5 (2001), 109–120. https://doi.org/10.1145/503271.503226

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. https://doi.org/10.4230/LIPIcs.ECOOP.2017.8

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. https://doi.org/10.1145/1889997.1890001

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. https://doi.org/10.1007/978-3-642-14107-2_24

Ivana Filipović, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010a. Abstraction for concurrent objects. *Theoretical Computer Science (TCS)* 411, 51–52 (2010), 4379–4398. https://doi.org/10.1016/j.tcs.2010.09.021

Ivana Filipović, 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. https://doi.org/10.1007/s00165-009-0125-8

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. https://doi.org/10.1145/3209108.3209174

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional proof-irrelevance without K. *PACMPL* 3, POPL (2019), 3:1–3:28. https://doi.org/10.1145/3290316

Abraham Ginzburg. 1968. *Algebraic Theory of Automata*. Academic Press. https://doi.org/10.1016/b978-1-4832-0016-3.50006-8

Alexey Gotsman and Hongseok Yang. 2012. Linearizability with ownership transfer. In *International Conference on Concurrency Theory (CONCUR)*. 256–271. https://doi.org/10.1007/978-3-642-32940-1_19

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. https://doi.org/10.1145/2676726.2676975

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. https://doi.org/10.1145/3192366.3192381

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. https://doi.org/10.1007/978-3-642-03741-2_6

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. https://doi.org/10.1145/1810479.1810540

Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2013. Aspect-oriented linearizability proofs. In *International Conference on Concurrency Theory (CONCUR)*. 242–256. https://doi.org/10.1007/978-3-642-40184-8_18

Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming*. M. Kaufmann. https://doi.org/10.1108/03684920810907904

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. https://doi.org/10.1145/78969.78972

C. A. R. Hoare. 1972. Towards a theory of parallel programming. In *Operating Systems Techniques*. 61–71. https://doi.org/10.1007/978-1-4612-6315-9_16

Bart Jacobs. 2016. *Introduction to coalgebra: towards mathematics of states and observation.* Cambridge University Press. https://doi.org/10.1017/cbo9781316823187

Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *ACM Symposium on Principles of Programming Languages (POPL)*. 271–282. https://doi.org/10.1145/1925844.1926417

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of the Third International Symposium on NASA Formal Methods (NFM 2011)*. https://doi.org/10.1007/978-3-642-20398-5_4

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. https://doi.org/10.1007/978-3-662-43951-7_19

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Comput. Syst.* 5, 4 (1983). https://doi.org/10.1145/69575.69577

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)* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In *ACM Symposium on Principles of Programming Languages (POPL)*. 637–650. https://doi.org/10.1145/2676726.2676980

Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. 2017. Proving linearizability using partial orders. In *European Symposium on Programming (ESOP)*. 639–667. https://doi.org/10.1007/978-3-662-54434-1_24

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. https://doi.org/10.1145/2429069.2429134

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. https://doi.org/10.1145/2499370.2462189

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. https://doi.org/10.1145/3158108

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. https://doi.org/10.1145/2103656.2103711

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. https://doi.org/10.1145/2603088.2603123

Nancy A. Lynch and Frits W. Vaandrager. 1995. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.* 121, 2 (1995), 214–233. https://doi.org/10.1006/inco.1995.1134

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. https://doi.org/10.1145/103727.103729

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: a verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS)*, Vol. 9583. 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Aleksandar Nanevski. 2016. Separation Logic and Concurrency. Lecture Notes for the Oregon Programming Languages Summer School (OPLSS). https://www.cs.uoregon.edu/research/summerschool/summer16/curriculum.php

Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2019a. Specifying Concurrent Programs in Separation Logic: Morphisms and Simulations. (2019). https://doi.org/10.5281/zenodo.3365991 Artifact including Coq source code.

Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2019b. Specifying Concurrent Programs in Separation Logic: Morphisms and Simulations. (2019). arXiv:1904.07136

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. https://doi.org/10.1007/978-3-642-54833-8_16

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and separation in Hoare Type Theory. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*. https://doi.org/10.1145/1159803.1159812

Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Dependent Types for Imperative Programs. In *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*

(ICFP 2008). https://doi.org/10.1145/1411204.1411237

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theoretical Computer Science (TCS)* 375, 1-3 (2007). https://doi.org/10.1016/j.tcs.2006.12.035

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Conference of the European Association for Computer Science Logic (CSL)*. 1–19. https://doi.org/10.1007/3-540-44802-0_1

Susan S. Owicki and David Gries. 1976. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM* 19, 5 (1976), 279–285. https://doi.org/10.1145/360051.360224

Frank Pfenning. 2001. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In *IEEE Symposium on Logic in Computer Science (LICS)*. 221–230. https://doi.org/10.1109/lics.2001.932499

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*. https://doi.org/10.1109/LICS.2002.1029817

J.J.M.M. Rutten. 2000. Universal coalgebra: a theory of systems. *Theoretical Computer Science* 249, 1 (2000), 3 – 80. https://doi.org/10.1016/s0304-3975(00)00056-6

Gerhard Schellhorn, Heike Wehrheim, and John Derrick. 2012. How to prove algorithms linearisable. In *Computer Aided Verification (CAV)*. 243–259. https://doi.org/10.1007/978-3-642-31424-7_21

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015a. Mechanized Verification of Fine-grained Concurrent Programs. In *ACM Conference on Programming Languages Design and Implementation (PLDI)*. https://doi.org/10.1145/2737924.2737964

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015b. Specifying and verifying concurrent algorithms with histories and subjectivity. In *European Symposium on Programming (ESOP)*. 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

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. https://doi.org/10.1145/3022671.2983999

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. https://doi.org/10.1145/3158116

Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *European Symposium on Programming (ESOP)*. 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

Kasper Svendsen, Lars Birkedal, and Aleksandar Nanevski. 2011. Partiality, State and Dependent Types. In *International Conference on Typed Lambda Calculi and Applications (TLCA)*. 198–212. https://doi.org/10.1007/978-3-642-21691-6_17

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. https://doi.org/10.1007/978-3-642-37036-6_11

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. https://doi.org/10.1145/2544174.2500600