Subjective Simulation as a Notion of Morphism for Composing Concurrent Resources

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

Recent approaches to verifying programs in separation logics for concurrency have used state transition systems (STSs) to specify the atomic operations of programs. A key challenge in the setting has been to compose such STSs into larger ones, while enabling programs specified under one STS to be lifted to a larger one, without reverification. This paper develops a notion of morphism between two STSs which permits such lifting. The morphisms are a constructive form of simulation between the STSs, and lead to a general and concise proof system. We illustrate the concept and its generality on several disparate examples, including staged construction of a readers/writers lock and its proof, and of proofs about quiescence when concurrent programs are executed without external interference.

1 INTRODUCTION

In many separation logics for shared-memory concurrent programs, a formal description of a concurrent resource takes a form of a state transition system (STS) [10, 16, 21]. The state space of an STS describes what holds of the resource's heap and auxiliary state at all times during execution, while the transitions specify the moves that programs operating over the resource are allowed to make atomically. Thus, resources are part of program specification: when verifying a program that operates over a resource, one not only has to establish the program's pre- and postcondition, but also show that the program respects the resource's state space and transitions. In the sequel, we use "resource" and "STS" interchangeably.[1]

One of the major challenges of the approach—which we address in this paper—has been to design a formalism for composing resources into new ones, which, moreover, allows the reuse of proofs carried out for programs written for constituent resources, as follows. Once resources are composed, it should be possible to *lift* a program that has been verified wrt. one of the component STSs, and automatically infer its correctness wrt. the composition, without any re-verification.

Consider the example of a concurrent resource in the style of Concurrent Separation Logic (CSL) [22]. This is a lock-protected shared heap satisfying a predicate, say I, (aka. resource invariant [23]) when no thread holds the lock. When the lock is acquired, the protected heap is transferred to the exclusive ownership of the acquiring thread. While in exclusive possession of the heap, the thread can modify the heap to temporarily violate I, but has to re-establish I before unlocking, when the heap becomes shared again.

CSL is coarse-grained, locking the whole data structure before modification. Nevertheless, it already illustrates the need for decomposition. A CSL-style resource performs two distinct functionalities: locking and unlocking on the one hand, and transferring heap ownership on the other. The two problems have separate concerns and can appear individually in different contexts. For example, transfer of heap ownership occurs when a concurrent stack operation allocates a new node in a private state, and then pushes it onto the shared stack, without actually locking the whole structure. Similarly, locking and unlocking may be considered independently of ownership

[1]Related works have also used names such as *concurrent protocols*, *distributed protocols*, and *concurroids* for similar concepts.

transfer, or in settings where the ownership discipline is more involved than in CSL. For example, in readers-writers lock [3, 6], when a reader acquires the lock, the protected heap is not transferred to the private ownership of that reader, but can be shared by all readers in the system. Thus, the two different functionalities are best formalized as individual STSs, which can then be composed into a CSL-style lock, or used separately.

However, to recover the CSL-lock functionality by composition, one must interconnect the states and transitions of the two components, as they are not independent. For example, let Spin be a resource implementing a spin lock. We will formally describe this resource in Section 2, but, as a first approximation, one may envision an STS with two states and two non-idle transitions, lock and unlock. Next, let Xfer be a resource implementing the ownership transfer of a heap, under resource invariant I. Again as an approximation, Xfer's states consist of a private and a shared heap, and the transitions move a set of pointers circumscribed by I between the two heaps. To reconstitute a CSL lock as a composition of Spin and Xfer, we have to ensure that whenever Spin transitions by taking the lock, Xfer is able to transfer the shared heap into private ownership of the locking thread: this heap must not already be privately owned. Dually, whenever Spin transitions to release the lock, then Xfer must ensure that there exists a chunk of private heap that satisfies invariant I and that can be transferred into the shared state. During either of these transitions by Spin, Xfer should not be able to perform any other manipulation of the heap, and vice versa.

Moreover, if we write a *program* over Spin, we should be able to lift it to operate on states that lie in the composition of Spin and Xfer. For example, a program for locking may be implemented as a loop trying to take a lock, until it succeeds. This program respects the transitions of Spin, because either it stays idle if it fails to take the lock, or it makes the lock transition of Spin in the loop's last iteration. Once this program is verified wrt. Spin, we should be able to *lift* it to work over the composition of Spin and Xfer, without additional proof obligations. Whenever the program would have taken a transition of Spin, the lifting has to take a transition in the composition, i.e., transform an Xfer part of the composed state by a specific, possibly non-idle, Xfer transition.

The customary mathematical structure for relating STSs are *simulations* [1]. However, most modern separation logics for concurrency, while using STSs to formalize resources, relate the resources, and tie them to program lifting, by notions other than simulations (see Section 6). Examples include *higher-order auxiliary code* [14, 27, 28], *atomicity tokens* [7, 16], and *protocol hooks* [11], among others. In practice, the use of each of these concepts leaves one with a sense that there is a simulation between underlying resources that is being implicitly constructed; but the simulation is never made an explicit object of the formalism.

In contrast, this paper advocates a form of simulation between STSs as a key concept to relate resources and formalize program lifting. If a resource V is a sub-component of W, as in the above example of Spin and CSL-style lock, then W simulates V. Then, a program e operating over V can easily be lifted to operate over W: whenever e takes a transition of V, the lifted program should take a corresponding transition of W, which is guaranteed to exist because of the simulation. The **fundamental contribution** of this paper is this notion of simulation as a foundation for separation logics for concurrency. Specifically, we develop a new logic which reformulates previous work on Fine-grained Concurrent Separation Logic (FCSL) [17, 21]. The new logic, also called FCSL, is designed around simulations to achieve significant conceptual and formal simplicity compared to the previous work on FCSL, or the other related works listed above. For example, we require only a *single inference rule* to reason about program lifting.

There are several hurdles to overcome in the design of FCSL, leading to the two main technical contributions of this paper. First, we must focus on a special kind of simulations, that are *constructive* in the sense of type theory. Whenever V can take a transition, it does not suffice merely to know that there *exists* a transition that W can take as well; we need a witness for the existential. Only then can we use our simulation as a *morphism* on programs, that is, a function that can modify a program over V on-the-fly, into a program over W. Our **first technical contribution** is to identify the properties that make a simulation be a morphism, in the above sense.

In more detail, the new FCSL Hoare triples have the form of a typing judgment $e : \{P\}\ A\ \{Q\}@V$. The judgment states that program e returns a value of type A (if it terminates), e respects the state space and transitions of V, and has precondition P and postcondition Q, assuming interference that also respects the state space and transitions of V. A morphism $f : V \longrightarrow W$ is a structure that relates the states of V and W, and maps the transitions of V to transitions of W. The following single inference rule lifts program e over V to program morph $f\ e$ over W by applying f to e:

$$\frac{e : \{P\}\ A\ \{Q\}@V}{\mathsf{morph}\ f\ e : \{f\char`\^P \mathbin{\boldsymbol{\wedge}} I\}\ A\ \{f\char`\^Q \mathbin{\boldsymbol{\wedge}} I\}@W}\ \textsc{Lift}$$

Intuitively, the behavior of morph $f\ e$ is to take the transition $f(t)$ in W, whenever e takes the transition t in V. And, $f\char`\^P$ is the action of $f$ on predicates over state, defined as $f\char`\^P = \lambda s_w.\, \exists s_v.\, (s_v, s_w) \in f \wedge P\ s_v$, where s_v and s_w are states from the state spaces of V and W, respectively.[2] I is a predicate over states of W, which is "preserved" by f in a sense that we formally define in Section 3.

Soundness considerations of the above rule lead to our **second technical contribution**, which is novel structure on resource transitions. In previous work on FCSL, a state of a resource distinguished between *self*-components (private to the specified thread), and *other*-components (private to the interfering threads). The *other*-component abstracted from the context of interfering threads, making it unnecessary to reverify programs when the number of interfering threads changed [17]. This state organization was named *subjective*, because it gave each thread its local (i.e., subjective) view of state ownership. In contrast, this paper extends the subjective dichotomy to transitions, and differentiates between *internal* and *external* transitions. The *internal* transitions of resource V are those that a program over V can take. The *external* transitions cannot be taken by a program directly, but they delimit how V can be combined with other resources, and in particular, how a thread over the combined resource can interfere with a thread over V. *External* transitions thus abstract from the resource context in which V appears, and serve as V's interface. A morphism $f : V \longrightarrow W$ is a simulation that treats *self* components and *internal* transitions differently from *other* components and *external* transitions, as follows.

(1) Every *internal* transition t of V is matched by an *internal* transition $f(t)$ of W, modifying *self*-components, but preserving *other*-components.
(2) Every *external* transition of W is matched by *one or more* transitions of V, of either kind, in succession, modifying *other*-components, but preserving *self*-components.

Requirement (1) ensures that morph $f\ e$ lifts the atomic steps of e from V to W. Requirement (2) ensures that atomic steps performed by interfering threads to morph $f\ e$ over W, can also be seen as atomic steps performed by interfering threads to e over V. Together, the requirements enable exploiting the Hoare type of e in the premiss of the Lift rule, and ensuring the latter's soundness.

The notion of morphism has applications that go beyond resource composition and lifting. For example, Section 3 shows how to add a new property I to the state space of a resource V, so long as I is inductive (i.e., preserved by V's transitions). Moreover, there is a *generic* morphism from V to the restricted resource V/I. Section 5 illustrates how to use morphisms in a generalized form of *indexed morphism families*, to formalize *quiescence* [21, 26]. This is a situation when a resource

[2] In separation logic, logical connectives operate on state predicates. Here, we make a typographic distinction between predicate connectives (bold font), and propositional connectives (regular font). For example, $P \mathbin{\boldsymbol{\wedge}} Q = \lambda s.\, P\ s \wedge Q\ s$.

V is installed in a private state of some program e. The children threads of e may compete for the new resource, but other threads cannot interfere, because they cannot access e's private state.

All our examples (including ones not discussed in the paper) and meta theory have been mechanized in Coq, and the sources are available in the supporting material.

2 OVERVIEW

We introduce FCSL by developing CSL-style locks in a decomposed manner. The resource Spin formalizes locking over the spin lock r. The resource Xfer formalizes ownership transfer of the protected heap, enforcing that a resource invariant I holds of the heap when it is shared. The resource CSL composes Spin and Xfer, enforcing that: (1) when Spin locks, Xfer enables the heap to be acquired by the locking thread, and (2) Spin unlocks only after Xfer has been placed in a state whereby I holds of the heap. A morphism $f : \mathsf{Spin} \to \mathsf{CSL}$ can lift Spin programs for locking and unlocking to CSL, thereby reusing the programs' code and proof in Spin.

2.1 Resource Spin for locking and unlocking

Physically, a spin lock is a Boolean pointer r, which is locked if r is true. Threads try to lock by executing CAS(r, false, true). The latter reads from r, and, if false, sets r to true, returning true to indicate successful locking. We assume that memory operations over a single pointer are atomic; thus, no threads can modify r between the reading and mutation by CAS. A thread that holds r, releases it by writing false into it. For verification, however, Spin cannot comprise only the boolean states indicating whether r is locked or not. It has to additionally track which thread, if any, actually holds r, as such threads will be allowed operations not allowed to others (e.g., unlocking). One way to track lock ownership is by thread id's, but we do not do so here. Instead, we endow Spin with a special form of *subjective* state (Section 1), described concretely below. As we shall see, subjective state will apply to all our resources, with uses well beyond replacing thread id's [17, 21].

Subjective states. We divide the state s of Spin into three components $s = (\mu_s, \pi, \mu_o)$. Each thread over Spin has these components in its name-space, but they may have different values in different threads. For example, the *self*-component μ_s equals own in the thread that holds the lock, but $\cancel{\text{own}}$ in all other threads. Dually, the *other*-component μ_o equals own in a thread whose environment holds the lock, and $\cancel{\text{own}}$ otherwise. The lock is taken if exactly one of μ_s and μ_o is own. Importantly, each thread is allowed to modify only its own μ_s value, but not μ_o, and dually, μ_s of one thread cannot be changed by others. This way, the division into *self* and *other* fields captures a form of ownership. On the other hand, the π component is under *joint* (i.e., shared) ownership. We introduce it with the view towards the composition of Spin and Xfer, and it is a Boolean indicating that the invariant I holds of the heap in Xfer. This heap is not part of Spin, so π is essentially a proxy that will be ascribed the explained meaning only after we compose Spin and Xfer. For now, it suffices to consider π as a field that a thread wanting to unlock r must set to true, in addition to having $\mu_s = \text{own}$.[3] In the sequel, we treat the field names as projections, and write, for example, $\mu_s(s)$ and $\mu_s(s')$, when we want to extract the first component of the states s and s', respectively.

The fields μ_s, μ_o, and π must be related by some conditions, which we describe next. First, we define the operation $\bullet$ on $O = \{\text{own}, \cancel{\text{own}}\}$ as follows: $x \bullet \cancel{\text{own}} = \cancel{\text{own}} \bullet x = x$ with $\text{own} \bullet \text{own}$ undefined. The operation is commutative, associative, with $\cancel{\text{own}}$ as the unit element, hence it endows O with the structure of a partial commutative monoid (PCM) [9, 16, 17, 21]. We can now abbreviate $\mu(s) = \mu_s(s) \bullet \mu_o(s)$ to capture the lock status; r is taken iff $\mu(s) = \text{own}$. Second, for

[3]It is customary in separation logic to refer to π as a *permission* to unlock. We refrain from doing so, as for us π is a necessary, but not sufficient condition for unlocking, as the thread must also set $\mu_s = \text{own}$.

each resource, we define its *flattening*, which maps the abstract state s into a heap $\ulcorner s \urcorner$, thereby declaring that the values μ_s, μ_o and π are *auxiliary* [20, 23]—they are introduced for verification, but do not matter in execution, where only $\ulcorner s \urcorner$ matters. Now we can define the state space of Spin, which relates μ_s, μ_o and π as follows.

$$
\begin{array}{rcl}
S(s) & \mathrel{\widehat{=}} & \mathsf{defined}\ (\mu(s)) \wedge r \neq \mathsf{null} \wedge (\mu(s) = \cancel{\mathsf{own}} \longrightarrow \pi(s)) \\
\ulcorner s \urcorner & \mathrel{\widehat{=}} & r \Mapsto (\mu(s) = \mathsf{own})
\end{array}
$$

The conjunct $\mathsf{defined}\ (\mu(s))$ encodes mutual exclusion: two different threads cannot simultaneously hold the lock because if $\mu_s(s) = \mu_o(s) = \mathsf{own}$, then $\mu(s)$ would be undefined. The conjunct $r \neq \mathsf{null}$ requires that r is a valid heap pointer. The last conjunct in $S(s)$ says that if the lock is free, then, in the eventual composition with Xfer, the protected heap of Xfer satisfies the invariant I, thus encoding the main property of CSL-style locking. The definition of $\ulcorner s \urcorner$ declares that Spin's physical heap contains only the lock r, which is locked if $\mu(s) = \mathsf{own}$.

Transitions. A transition is a binary relation between a pre-state s and post-state s', formalizing the atomic operations of a resource. In the display below, we present the transitions of Spin, where we assume that both states s and s' satisfy Spin's S.

$$
\begin{array}{rcl}
\mathsf{lock\_tr}\ s\ s' & \mathrel{\widehat{=}} & \mu(s) = \cancel{\mathsf{own}} \wedge \mu_s(s') = \mathsf{own} \wedge \pi(s') \\
\mathsf{unlock\_tr}\ s\ s' & \mathrel{\widehat{=}} & \mu_s(s) = \mathsf{own} \wedge \pi(s) \wedge \mu_s(s') = \cancel{\mathsf{own}} \\
\mathsf{set\_tr}\ b\ s\ s' & \mathrel{\widehat{=}} & \mu_s(s) = \mu_s(s') = \mathsf{own} \wedge \pi(s') = b \\
\mathsf{id\_tr}\ P\ s\ s' & \mathrel{\widehat{=}} & P\ s \wedge s' = s
\end{array}
$$

Transition $\mathsf{lock\_tr}$ describes a *successful* acquisition of the lock. It can be taken only if the lock is free ($\mu(s) = \cancel{\mathsf{own}}$), and in the post-state, the lock is held by the acquiring thread ($\mu_s(s') = \mathsf{own}$). By definition of S, π must be set in s, and it remains so in s'. On the other hand, $\mathsf{set\_tr}$ takes a boolean b as an input, and sets π to b. It can be performed only by a thread that holds the lock ($\mu_s(s) = \mathsf{own}$). Similar explanation applies to $\mathsf{unlock\_tr}$ which describes unlocking. Notice that transitions may modify μ_s and π, but can only read μ_o, as the latter is owned by other threads. It is therefore always the case in a transition that $\mu_o(s') = \mu_o(s)$, which we thus assume as default, and omit stating explicitly. The idle transition $\mathsf{id\_tr}$ is taken by a thread when it executes no state changes, i.e., it stays idle. We parametrize $\mathsf{id\_tr}$ by a predicate P, to describe what holds of the pre-state when the transition is taken. As we show promptly, this will be exploited when defining the *action* for locking, when the idle transition will describe when the locking *fails*. If P is the always-true predicate, we omit it.

Actions. Transitions describe the steps of a resource at the level of specification, while *actions* describe the atomic operations *at the level of programs*. Actions are composed out of one or more transitions, and return a result that identifies the transition taken by the action. Thus, an action is a relation between the output result, the input state, and the output state. For example, the action $\mathsf{trylock\_act}$ takes the transition $\mathsf{lock\_tr}$ in the case of successful locking, and $\mathsf{id\_tr}\ P$ otherwise. We use $P \mathrel{\widehat{=}} \lambda s.\, \mu(s) = \mathsf{own}$ to indicate that the locking fails only if the lock were taken in s.

$$
\mathsf{trylock\_act}\ (b : \mathit{bool})\ s\ s' \mathrel{\widehat{=}} \mathsf{if}\ b\ \mathsf{then}\ \mathsf{lock\_tr}\ s\ s'\ \mathsf{else}\ \mathsf{id\_tr}\ (\lambda s.\, \mu(s) = \mathsf{own})\ s\ s'.
$$

While $\mathsf{trylock\_act}$ is defined over the whole state of Spin, including auxiliary values such as $\mu(s)$, notice that when the state is flattened to the pointer r, the action, intuitively, behaves like $\mathsf{CAS}(r, \mathsf{false}, \mathsf{true})$ discussed before. We say that $\mathsf{trylock\_act}$ *erases* to CAS, or alternatively, that $\mathsf{trylock\_act}$ annotates CAS with *auxiliary code* for updating μ_s, μ_o and π. All our actions erase to some memory operation that executes atomically on hardware.

The action unlock_act does not branch, but takes the unlock_tr transition, returning the result of unit type. The action unlock_act erases to the atomic operation of writing false into r.

$$\text{unlock\_act}\ (x : \mathit{unit})\ s\ s' \ \widehat{=}\ \text{unlock\_tr}\ s\ s'$$

We can now implement the programs for locking and unlocking r.[4] The former loops executing trylock_act until it succeeds to acquire r, while the latter just invokes unlock_act.

lock : $\{\lambda s.\, \top\}\ \{\lambda s.\, \mu_s(s) = \text{own} \wedge \pi(s)\}$@Spin =
  do ($b \leftarrow$ atomic trylock_act;
    if b then ret () else lock)

unlock : $\{\lambda s.\, \mu_s(s) = \text{own} \wedge \pi(s)\}$
  $\{\lambda s.\, \mu_s(s) = \not\text{own}\}$@Spin =
  do (atomic unlock_act)

The precondition of lock is $\top$, hence lock can be invoked in any state. The postcondition indicates that the lock is acquired by the invoking thread, and π is set. This holds because the program loops, until it manages to execute lock_tr, which terminates with the lock acquired and π set. The precondition of unlock requires the invoking thread to hold the lock, and π to be set. Upon termination, the thread does not have the lock anymore, as expected, but also notice that π is undetermined. Unlock_tr terminates with π set, thus, immediately upon execution of unlock, we know that π will be set. However, our specifications state only *stable* properties of state, i.e., those that remain invariant under interference of other threads over Spin. In this particular case, another thread may reset π after unlock terminates, which is why π is undetermined in unlock's postcondition. On the other hand, π holds stably in lock's postcondition because only the thread holding the lock can reset π.

2.2 Resource Xfer for heap ownership transfer

A state s of Xfer has the form $s = (\sigma_s, (\sigma_j, v), \sigma_o)$. The fields σ_s and σ_o describe the private heaps of the thread operating over Xfer, and the thread's environment, respectively. The field σ_j is the shared heap on which we consider the satisfaction of the resource invariant I. Heaps form a PCM under the operation of disjoint union, with empty as unit, just as was the case with the *self* and *other* fields in Spin; we abbreviate the total heap of s as $\sigma(s) = \sigma_s(s) \bullet \sigma_j(s) \bullet \sigma_o(s)$. The field v is a boolean indicating the satisfaction of the invariant. The state space of Xfer is defined as follows.

$$\begin{array}{rcl} S(s) & = & \text{defined}\ (\sigma(s)) \wedge \text{if}\ v(s)\ \text{then}\ I\ \sigma_j(s)\ \text{else}\ \sigma_j(s) = \text{empty} \\ \ulcorner s \urcorner & = & \sigma(s) \end{array}$$

Specifically, if v is true, then I holds of σ_j. Otherwise, the contents of σ_j have been transferred to σ_s of some thread, and thus σ_j equals empty heap.

The transitions of Xfer describe the exchange of heaps between σ_j and σ_s. We name them close_tr and open_tr, because they close and open the invariant I for violation, by moving a heap satisfying I into and out of σ_j.

$$\begin{array}{rcl} \text{close\_tr}\ s\ s' & \widehat{=} & \exists h.\, \sigma_s(s) = \sigma_s(s') \bullet h \wedge I\ h \wedge \neg v(s) \wedge \sigma_j(s') = h \wedge v(s') \\ \text{open\_tr}\ s\ s' & \widehat{=} & v(s) \wedge \sigma_s(s') = \sigma_s(s) \bullet \sigma_j(s) \wedge \sigma_j(s') = \text{empty} \wedge \neg v(s') \end{array}$$

Close_tr moves the subheap h of $\sigma_s(s)$ into $\sigma_j(s')$. The moved heap h must satisfy I, as otherwise, s' will not satisfy S. The transition sets $v(s')$ to indicate the satisfaction of I in s'. Symmetrically, open_tr moves $\sigma_j(s)$ into $\sigma_s(s')$, thereby leaving $\sigma_j(s') = \text{empty}$. We elide here the few additional Xfer transitions, such as id_tr P (defined identically as in Spin), and the transitions for mutating, allocating, and deallocating pointers in σ_s, as they are not essential for our present goal of explaining resource composition and morphisms.

[4] The proofs of the type ascriptions are in our Coq files.

2.3 Composing Spin and Xfer into CSL

The resource CSL combines the functionalities of Spin and Xfer, and admits morphisms from both. Specifically, the morphism from Spin will allow us to automatically lift lock and unlock to CSL.

A state of CSL pairs up the states of Spin and Xfer, point-wise in the *self*, *joint* and *other* components. In other words, $s = ((\mu_s, \sigma_s), (\pi, (\sigma_j, \nu)), (\mu_o, \sigma_o))$. We write $s\backslash 1$ (resp. $s\backslash 2$) for the first (resp. second) point-wise projection of s. Thus, $s\backslash 1 = (\mu_s, \pi, \mu_o)$ is a state of Spin, and $s\backslash 2 = (\sigma_s, (\sigma_j, \nu), \sigma_o)$ is a state of Xfer. We exclude some state pairings, however, as the following definitions indicate:

$$
\begin{array}{rcl}
S(s) & = & \mathsf{Spin}.S(s\backslash 1) \wedge \mathsf{Xfer}.S(s\backslash 2) \wedge \mathsf{defined} \ulcorner s \urcorner \wedge \pi(s) = \nu(s) \\
\ulcorner s \urcorner & = & \mathsf{Spin}.\ulcorner s\backslash 1 \urcorner \bullet \mathsf{Xfer}.\ulcorner s\backslash 2 \urcorner
\end{array}
$$

In particular, we require that: (1) The paired states have disjoint heaps, i.e. the lock r from Spin does not occur as a pointer in $\sigma(s)$ in Xfer. This is imposed by the conjunct defined $\ulcorner s \urcorner$; (2) The booleans π and ν from the component STSs must be equal in the composition. This provides π with the intended semantics from Section 2.1, whereby it allows unlocking only if the protected heap satisfies I. Indeed, when $\pi(s) = \nu(s) = \mathsf{true}$, then $I\ \sigma_j(s)$ by definition of Xfer.S, and Spin can invoke unlock_tr. Dually, when $\pi(s) = \nu(s) = \mathsf{false}$, then $\sigma_j(s) = \mathsf{empty}$, as the protected heap is in private ownership of the locking thread, where I may be violated. Correspondingly, Spin cannot invoke unlock_tr. However, the states where $\pi(s) \neq \nu(s)$ are of no interest, and are ruled out by S.

Transitions of CSL combine the transitions of Spin and Xfer, as follows, omitting id_tr for brevity:

$$
\begin{array}{rcl}
\mathsf{lock\_tr} & = & \mathsf{Spin.lock\_tr} * \mathsf{Xfer.id\_tr} \\
\mathsf{unlock\_tr} & = & \mathsf{Spin.unlock\_tr} * \mathsf{Xfer.id\_tr} \\
\mathsf{close\_tr} & = & \mathsf{Spin.set\_tr(true)} * \mathsf{Xfer.close\_tr} \\
\mathsf{open\_tr} & = & \mathsf{Spin.set\_tr(false)} * \mathsf{Xfer.open\_tr}
\end{array}
$$

We formally define the operation $t_1 * t_2$ of *coupling* of transitions in Section 3, but for now it suffices to say that $t_1 * t_2$ *simultaneously* takes t_1 over $s\backslash 1$ (a state of Spin), and t_2 over $s\backslash 2$ (a state of Xfer). Thus, lock_tr performs the lock transition of Spin, while remaining idle on Xfer, and similarly for unlock_tr. On the other hand, open_tr (and close_tr is similar) executes Xfer.open_tr to transfer the shared heap to private ownership, resetting $\nu(s)$ in the process. Spin.set_tr(false) has to be simultaneously executed, in order to maintain $\pi(s) = \nu(s)$.

2.4 Morphisms

We next construct the morphism $f : \mathsf{Spin} \to \mathsf{CSL}$ that will allow us to lift lock and unlock (Section 2.1) from Spin to CSL, thereby reusing their Spin implementation and proof. The morphism consists of two parts: a relation on the states of Spin and CSL, and a function mapping the transitions of Spin to those of CSL. Given state s of Spin and s' of CSL, the state-relation part of f is:

$$(s, s') \in f \;\widehat{=}\; s = s'\backslash 1,$$

using that a CSL-state is a pair of a Spin and Xfer state. The transition-map part of f is defined as:

$$
\begin{array}{rcl}
f(\mathsf{Spin.lock\_tr}) & \widehat{=} & \mathsf{CSL.lock\_tr} \\
f(\mathsf{Spin.unlock\_tr}) & \widehat{=} & \mathsf{CSL.unlock\_tr} \\
f(\mathsf{Spin.id\_tr}\ P) & \widehat{=} & \mathsf{CSL.id\_tr}\ (\lambda s.\, P\ s\backslash 1) \\
f(\mathsf{Spin.set\_tr}\ b) & \widehat{=} & \mathsf{undefined}
\end{array}
$$

The key role of f is to establish a simulation between Spin and CSL, i.e., whenever Spin takes a transition t, CSL can take a transition $f(t)$, with the input states of t and $f(t)$ being related by the state-relation of f, and similarly for the output states. When $t \in \{\text{Spin.lock\_tr}, \text{Spin.unlock\_tr}, \text{Spin.id\_tr}\ P\}$, it is easy to see that this property holds. For example, if $t = \text{Spin.lock\_tr}$, then $f(t) = \text{CSL.lock\_tr} = \text{Spin.lock\_tr} * \text{Xfer.id\_tr}$. When t can be taken in Spin, clearly $f(t)$ can be taken in CSL, since Xfer.id_tr does not impose any additional constrains.

Importantly, it is *not possible* to make this property hold for $t = \text{Spin.set\_tr}$. We could consider defining f on t as, e.g., $f(\text{Spin.set\_tr (true)}) = \text{CSL.close\_tr} = \text{Spin.set\_tr(true)} * \text{Xfer.close\_tr}$, but such a definition does not give a simulation. Namely, it is *not* the case that when Spin.set_tr($true$), then Xfer.close_tr can follow, as the latter requires a further condition that there exist subheap h of $\sigma_s(s)$ such that $I\ h$ holds. The existence of h is not guaranteed by Spin.set_tr(true).

This motivates our *division* of transitions into *internal* and *external*, whereby morphisms are defined only on the *internal* ones. For Spin, the *internal* transitions are Spin.lock_tr, Spin.unlock_tr and Spin.id_tr, and the *external* transition is Spin.set_tr, on which f remains undefined. Intuitively, *external* transitions are "incomplete" operations, to be "completed" by the outside world, to which the *external* transitions are an interface. For example, Spin.set_tr is *external*, because the very role of π, which this transition manipulates, is to tie Spin to another resource, in this case Xfer. In the case of Xfer, we similarly classify close_tr and open_tr as *external*, as they too are incomplete, but for a somewhat different reason. Namely, an action involving these transitions cannot be ascribed a stable Hoare triple in and of itself. Indeed, a program trying to perform Xfer.open_tr cannot rely that $\nu(s)$ holds—and thus that there is a heap in the shared state to be moved—as another simultaneous thread may acquire the heap and reset $\nu(s)$. This is avoided in CSL.open_tr, which couples Xfer.open_tr with Spin.set_tr(false), and can thus be executed only by a thread holding the lock. Hence, in CSL, open_tr and similarly close_tr, are *internal*. [5]

Since we want morphisms to act on programs such as lock and unlock in Section 2.1, the actions that a program takes must be composed of *internal* transitions only. For example, programs lock and unlock use actions trylock_act and unlock_act, which are themselves defined in terms of Spin transitions lock_tr, unlock_tr and id_tr, but not set_tr. We can thus lift lock and unlock to CSL, by applying the Lift rule with morphisms f and $I\ \widehat{=}\ \lambda s.\, \sigma_s(s) = h$.

$$
\begin{aligned}
\mathsf{lock}' \;&:\; [h].\, \{\lambda s.\, \sigma_s(s) = h\}\ \{\lambda s.\, \mu_s(s) = \mathsf{own} \wedge \nu(s) \wedge \sigma_s(s) = h\}@\mathsf{CSL} = \\
&\qquad \mathsf{do}\ (\mathsf{morph}\ f\ \mathsf{lock}) \\
\mathsf{unlock}' \;&:\; [h].\, \{\lambda s.\, \mu_s(s) = \mathsf{own} \wedge \nu(s) \wedge \sigma_s(s) = h\}\ \{\lambda s.\, \mu_s(s) = \not{\mathsf{own}} \wedge \sigma_s(s) = h\}@\mathsf{CSL} = \\
&\qquad \mathsf{do}\ (\mathsf{morph}\ f\ \mathsf{unlock})
\end{aligned}
$$

The operational intuition behind lock′ (and unlock′ is similar) is that it executes lock, modifying lock's transitions by f. Program lock loops executing Spin.id_tr, until it finally executes Spin.lock_tr. Accordingly, lock′ will keep executing CSL.id_tr until it finally executes CSL.lock_tr, the latter merely extending Spin.lock_tr with Xfer.id_tr. Thus, the specification of lock′ is similar to that of lock in that it describes the modification to μ_s, but here it also states that the private heap $\sigma_s(s)$ is unchanged from the precondition to the postcondition, as in both, it equals the bound variable h. The latter could not have been specified for lock, because the field σ_s is not part of Spin, but is added by Xfer. In lock′ we use $\nu(s)$ instead of $\pi(s)$, as the two are equal by the definition of CSL's state space. In CSL we can further ascribe stable specification to close_tr and open_tr, since

[5]It is possible to make Xfer.open_tr stable, and thus internal, by introducing an additional field of type O that tracks if a thread can execute the transition, and an additional external transition to manipulate the extra field. For simplicity, we do not explore such design here, but it is not precluded by the system.

these are now *internal* transitions.

$$\begin{array}{lll} \mathsf{close} & : & [h_1].\,\{\lambda s.\,\exists h_2.\,\mu_s(s) = \mathsf{own} \wedge \neg\nu(s) \wedge \sigma_s(s) = h_1 \bullet h_2 \wedge I\ h_2\} \\ & & \qquad \{\lambda s.\,\mu_s(s) = \mathsf{own} \wedge \nu(s) \wedge \sigma_s(s) = h_1\}@\mathsf{CSL} = \\ & & \mathsf{do}\ (\mathsf{atomic}\ (\lambda x : \mathit{unit}.\,\mathsf{close\_tr})) \\ \mathsf{open} & : & [h_1].\,\{\lambda s.\,\mu_s(s) = \mathsf{own} \wedge \nu(s) \wedge \sigma_s(s) = h_1\} \\ & & \qquad \{\lambda s.\,\exists h_2.\,\mu_s(s) = \mathsf{own} \wedge \neg\nu(s) \wedge \sigma_s(s) = h_1 \bullet h_2 \wedge I\ h_2\}@\mathsf{CSL} = \\ & & \mathsf{do}\ (\mathsf{atomic}\ (\lambda x : \mathit{unit}.\,\mathsf{open\_tr})) \end{array}$$

We can then sequentially compose lock′; open and close; unlock′, to obtain programs that combine lock operations with ownership transfer.

2.5 Dividing Xfer into Shar and Priv

It is very useful to further subdivide Xfer into two components Shar and Priv, which separately deal with shared heaps and private heaps, respectively, and then inject each by means of a morphism into Xfer. Shar contains the fields σ_j and ν, while Priv contains σ_s and σ_o. Both have their own copies of give_tr and trans_tr transitions which are parametrized by the heap h. In the case of Shar (resp. Priv), these transitions describe how h can be taken out of σ_j (resp. σ_s) or into it, but do not specify from which resource h is received, or to which resource it is given away. Clearly, because they describe interaction with the unspecified outside world, these transitions must be *external*.

$$\begin{array}{rcl} \mathsf{Shar.take\_tr}\ h\ s\ s' & \mathrel{\widehat{=}} & I\ h \wedge \neg\nu(s) \wedge \sigma_j(s') = h \wedge \nu(s') \\ \mathsf{Shar.give\_tr}\ h\ s\ s' & \mathrel{\widehat{=}} & h = \sigma_j(s) \wedge \nu(s) \wedge \sigma_j(s') = \mathsf{empty} \wedge \neg\nu(s') \\ \mathsf{Priv.take\_tr}\ h\ s\ s' & \mathrel{\widehat{=}} & \sigma_s(s') = h \bullet \sigma_s(s) \\ \mathsf{Priv.give\_tr}\ h\ s\ s' & \mathrel{\widehat{=}} & \sigma_s(s) = h \bullet \sigma_s(s') \end{array}$$

Dividing the functionality of Xfer will allow us to transfer the shared heap σ_j of Shar to some resource other than Priv. We will exploit this subdivision in Section 4 on readers/writers, to facilitate reuse when formalizing different heap ownership modes (i.e., heap owned by a writer vs. heap owned by readers).

3 FORMAL STRUCTURES

3.1 Definitions

Definition 3.1 (State-type and state). A *state-type* is a pair (U, T) of a PCM U and a type T. A *state* of state-type (U, T) is a triple $s = (a_s, a_j, a_o)$ of type $U \times T \times U$. We use the labels as projections out of s. The projections $a_s(s)$ and $a_o(s)$ of type U are called *self* and *other* component, respectively. The projection $a_j(s)$ of type T is called *joint* component. The *self* component holds the values that are private to the specified thread, and cannot be changed by other threads. Dually, *other* component holds the values that are private to the environment of the specified thread, and cannot be changed by the specified thread. The *joint* component holds the value that can be changed by every thread.

In a specific resource, we name the components with a resource-specific name, but use a_s, a_j, a_o when we discuss resources in general. The $a_s(s)$ and $a_o(s)$ components of a state s present the local view of a thread that operates on s. Different threads operating simultaneously on the same resource may have different values for the a_s and a_o components of their states, depending on the operations that they have completed. For example, in Section 2, a thread that acquired the lock will have $a_s(s) = \mu_s(s) = \mathsf{own}$, whereas a thread not holding the lock will have $a_s(s) = \mu_s(s) = \overline{\mathsf{own}}$. If these threads execute at the same time, we further know that in the first thread $a_o(s) = \mu_o(s) = \overline{\mathsf{own}}$ and in the second, $a_o(s) = \mu_o(s) = \mathsf{own}$. In general, given any thread and a state s, the view of the whole concurrent environment (i.e. all of the threads concurrent to the considered thread), can be obtained by *transposition* of s, as per the following definition.

Definition 3.2 (State transposition). Given a state $s = (a_s, a_j, a_o)$, the *transposition* of s is the state $s^\top = (a_o, a_j, a_s)$.

As customary in separation logic, a common operation in FCSL is that of *framing*, i.e., adding values to state components. In FCSL, we consider framing of both of the PCM-valued components.

Definition 3.3 (Two notions of framing). Let $p \in U$ and s be a state of state-type (U, T). The *self-framing of s with p* is the state $s \lhd p = (a_s(s) \bullet p, a_j(s), a_o(s))$. Dually, *other-framing of s with p* is $s \rhd p = (a_s(s), a_j(s), p \bullet a_o(s))$.

A predicate is global if it is independent of the framing direction.

Definition 3.4 (Globality). Predicate P over states of state-type (U, T) is *global* if $P(s \lhd p) \leftrightarrow P(s \rhd p)$.

Using again the notation from Section 2, an example of a global predicate is $P(s) \mathrel{\widehat{=}} \mu(s) = \mathsf{own}$. By constraining the combined value $\mu(s) = \mu_s(s) \bullet \mu_o(s)$, P says that the lock is taken, but elides saying by whom. This is a general property; a global predicate P depends only on the combination $a_s(s) \bullet a_o(s)$, but not on the individual values of $a_s(s)$ and $a_o(s)$. Indeed, by definition, if P is global, then $P(a_s, a_j, a_o) \leftrightarrow P(a_s \bullet a_o, a_j, 1_U) \leftrightarrow P(1_U, a_j, a_s \bullet a_o)$, where 1_U is the unit of the PCM U. Thus, while a_s and a_o capture the effect on the resource by the specified thread and by the concurrent environment, respectively, a global predicate captures the total effect of all the threads, ignoring which thread did exactly what.

Next, we define the properties of a resource state space. For example, these will are satisfied by state spaces of Spin, Xfer and CSL from Section 2.

Definition 3.5 (State space). State space S of state-type (U, T) is a predicate over states (equivalently, set of states) of state type (U, T), that satisfies the following properties:

(1) *(validity)* if $S(s)$ then defined $(a_s(s) \bullet a_o(s))$
(2) S is global

Condition (1) in Definition 3.5 captures that we are only interested in states where the current thread and its concurrent environment have jointly performed a valid effect over the resource. For example, on Section 2, this condition imposes that we cannot have $\mu_s(s) = \mu_o(s) = \mathsf{own}$, i.e., the lock cannot be simultaneously held by a thread and by its environment. The globality condition (2) closes up the state-space under local views of simultaneous threads. If two states s_1 and s_2 are such that $a_j(s_1) = a_j(s_2)$ and $a_s(s_1) \bullet a_o(s_1) = a_s(s_2) \bullet a_o(s_2)$, then s_1 and s_2 represent the same moment in time of the resource, but from the point of view of two different concurrent threads. S being global means that S contains either both or neither of s_1 and s_2.

Definition 3.6 (Flattening). Let S be a state space of state-type (U, T). *Flattening* $\ulcorner - \urcorner : S \longrightarrow \mathsf{heap}$ is a function satisfying the following properties.

(1) if $S(s)$ then defined $\ulcorner s \urcorner$
(2) $\ulcorner s \lhd p \urcorner = \ulcorner s \rhd p \urcorner$

When we want to emphasize the state space S, we write $S.\ulcorner s \urcorner$ instead of $\ulcorner s \urcorner$.

Similarly to Definition 3.5, condition (1) captures that we only track resources whose flattened heap is valid, i.e., it does not contain the null pointer, or duplicate pointers. Condition (2) is similar to globality of S, and says that flattening is independent of thread-local views.

Definition 3.7 (State product). Let s_i be states of state-types (U_i, T_i), $i = 1, 2$. The product state $[s_1, s_2]$ defined as

$$[s_1, s_2] \mathrel{\widehat{=}} ((a_s(s_1), a_s(s_2)), (a_j(s_1), a_j(s_2)), (a_o(s_1), a_o(s_2))$$

is of state-type $(U_1 \times U_2, T_1 \times T_2)$, where $U_1 \times U_2$ is a PCM with join and unit defined point-wise. Symmetrically, given a state s of state-type $(U_1 \times U_2, T_1 \times T_2)$, the state $s\backslash i$ defined as $(\pi_i(a_s(s)), \pi_i(a_j(s)), \pi_i(a_o(s))$ is of state-type (U_i, T_i), $i = 1, 2$. The usual beta and eta laws for products hold, i.e.: $[s_1, s_2]\backslash i = s_i$ and $s = [s\backslash 1, s\backslash 2]$.

Definition 3.8 (State space product). Let S_i be a state space of state-type (U_i, T_i), $i = 1, 2$. Then the following define a valid state space and flattening over the product states:

$$\begin{array}{rcl} (S_1 \times S_2)\ s & \mathrel{\widehat{=}} & S_1(s\backslash 1) \wedge S_2(s\backslash 2) \wedge \text{defined} \ulcorner s \urcorner \\ \ulcorner s \urcorner & \mathrel{\widehat{=}} & S_1.\ulcorner s\backslash 1 \urcorner \bullet S_2.\ulcorner s\backslash 2 \urcorner \end{array}$$

The conjunct defined $\ulcorner s \urcorner$ imposes that the flattened heaps of component states are disjoint, in order to satisfy the requirement of Definition 3.6.(1).

Definition 3.9 (Transition). Let S be a state space of state-type (U, T). *Transition* t over S is a binary relation on states, satisfying the following properties.

(1) *(functionality)* 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) *(locality)* if $t\ (s \rhd p)\ s'$ then there exists s'' such that $s' = s'' \rhd p$ and $t\ (s \lhd p)\ (s'' \lhd p)$
(4) *(S-preservation)* if $t\ s\ s'$ and $S(s)$ then $S(s')$

When we want to emphasize the state space S wrt. which the transition is defined, we write $S.t$ instead of t, and refer to t as an S-transition. We say that a state s is *safe* for a transition t, if there exists s' such that $t\ s\ s'$.

Functionality requires that transitions are partial functions: the output state of a transition may be undefined on some input state, but if defined, it is unique. Thus, transitions are deterministic operations. This includes allocation, which separation logics often model non-deterministically. In our Coq files, we implement a simple concurrent allocator as a resource which keeps a free list, abstract from the clients. The allocator deterministically models allocation and deallocation by interacting with clients via transitions that transfer the head pointer of the free list back and forth, much like Xfer resource in Section 2 transferred a heap between private and joint state.

Other-fixity captures that transitions cannot change the other-view a_o of a thread, which are read-only, as already illustrated in Section 2.

Locality is a form of frame property from Abstract Separation Logic (ASL) [5]. Let $s = (a_s, a_j, a_o)$, and $s' = (a'_s, a'_j, a'_o)$, and assume that $t\ (s \rhd p)\ s'$. Ignoring *joint* and *other* components for a moment, the assumption says that executing t in a state with the *self* component a_s results in a state with the *self* component a'_s. The locality property says that if we increase the input *self*-component to $a_s \bullet p$, then the result and the increment are preserved; that is, the output *self* component is $a'_s \bullet p$. The specific of FCSL, compared to ASL, or other separation logics, is that the assumption $t\ (s \rhd p)\ s'$ requires the frame p to be available in the *other* component of the input state. In this sense, locality is a property stating an invariance of transitions under a realignment of local views of threads, whereby we take a portion p of the "effect" ascribed to an environment thread, and assign p to the specified thread.

Finally, the S-preservation property states that transitions preserve the state space. We have tacitly assumed this property in the examples in Section 2.

Definition 3.10 (Transition coupling). Let t_i be an S_i-transition, $i = 1, 2$. Then *coupling of* t_1 *and* t_2 is the $(S_1 \times S_2)$-transition $t_1 * t_2$, defined as:

$$(t_1 * t_2)\ s\ s' \mathrel{\widehat{=}} t_1\ (s\backslash 1)\ (s'\backslash 1) \wedge t_2\ (s\backslash 2)\ (s'\backslash 2) \wedge \text{defined} \ulcorner s' \urcorner$$

The coupled transition $t_1 * t_2$ executes t_1 and t_2 simultaneously, each on its respective portion of the input state. By the properties of $S_1 \times S_2$, we can assume that the input state s will have a valid flattening, i.e., that the heaps $\ulcorner s\backslash 1 \urcorner$ and $\ulcorner s\backslash 2 \urcorner$ are disjoint. However, when t_1 and t_2 transition *individually*, they might produce respective ending states that share a common pointer (e.g., t_1 and t_2 may receive the same pointer from the allocator). The conjunct defined $\ulcorner s' \urcorner$ prevents the coupled transition from ever synchronizing t_1 and t_2 in such a way.

Definition 3.11 (Internal transition). An S-transition t is *internal* if it preserves the heap domain of its input and output state; that is, whenever $t\ s\ s'$ then $\ulcorner s \urcorner$ and $\ulcorner s' \urcorner$ contain the same pointers.

Internal transitions are important because, intuitively, the set of their safe states is not affected by coupling with other internal transitions. More formally, if s_1, s_2 are safe for (internal) t_1, t_2, respectively, and $\ulcorner s_1 \urcorner$ is disjoint from $\ulcorner s_2 \urcorner$, then by Definition 3.11, $[s_1, s_2]$ is safe for $t_1 * t_2$. We build atomic actions of programs out of internal transitions only. Thus, the safety of a program whose atomic actions utilize the internal transition t_1 will not be affected if t_1 is coupled with an internal action t_2 over a disjoint state space. This property is hence key for soundly lifting a program over one resource, say Spin, to a combined resource, say CSL, which couples the transitions of Spin with those of Xfer.

External transitions are not required to preserve heap domains. External transitions describe interaction with other resources, and enlarging or shrinking a resource's heap is a form of interaction. For example, the transitions take_tr and give_tr from Section 2.5, acquire a new heap, or give away a part of the existing heap, respectively. External transitions cannot be used to build actions *directly*, but external transitions of different resources can be coupled into an internal transition of a combined resource, and then used in actions. For example, coupling Shar.give_tr h and Priv.take_tr h in Section 2.5, produces an effect of moving the heap h from Shar to Priv. But in the combination Xfer of Shar and Priv, this move is an internal effect overall, essentially corresponding to the internal transition Xfer.open_tr.

Definition 3.12 (Resource). A *resource* (or *STS*) is a tuple $V = (U, T, S, \Delta_i, \Delta_e)$, where S is a state space of state-type (U, T), and Δ_i and Δ_e are sets of internal and external S-transitions, respectively. We let $\Delta = \Delta_i \cup \Delta_e$ denote the set of all transitions. When V's components are not explicitly named, we refer to them using the dot-notation. That is, $V.U$ is V's PCM, $V.T$ is V's type, etc. A state s is a V-state, if it is of state-type $(V.U, V.T)$.

Definition 3.13 (Inductivity). Let V be a resource, and I a predicate over V-states. We say that I is an inductive invariant for V, or V-inductive for short, if it is preserved by the internal transitions of V; that is:

- for every $t \in V.\Delta_i$, if $t\ s\ s'$ and $I\ s$ then $I\ s'$.

Definition 3.14 (Other-stepping). Let V be a resource and s, s' be V-states. We say that s *other*-steps by V to s', written $s \underset{V}{\longrightarrow} s'$, if there exists a transition $t \in V.\Delta$ (thus, either internal or external) such that $t\ s^\top\ s'^\top$. We write $\underset{V}{\longrightarrow}^*$ for reflexive-transitive closure of $\underset{V}{\longrightarrow}$.

Because Definition 3.14 uses transpositions of s and s', the relation $s \underset{V}{\longrightarrow}^* s'$ expresses, from the point of view of the specified thread, that s can be modified into s' by the actions of the interfering threads. Other-stepping admits all transitions in $V.\Delta$, not only the internal ones. We include the external transitions to account for the possibility that a resource can be modified by interfering programs that operate not over V, but over some extension of V. For example, a heap in Priv may be augmented with another heap h acquired from Shar, once Priv and Shar are combined into Xfer.

Definition 3.15 (Stability). Let V be a resource. Predicate P over V-states is *stable in state* s if whenever $s \xrightarrow[V]{}^{*} s'$, then $P\ s'$. P is *stable* if it is stable in state s, for every s for which $P\ s$. Given P, we define its *stabilization* $P^{\bullet}$ as $P^{\bullet}\ s\ \widehat{=}\ \forall s'.\, s \xrightarrow[V]{}^{*} s' \to P\ s'$. It is easy to see that $P^{\bullet}$ is stable, and that P is stable iff $\forall s.\, P\ s \to P^{\bullet}\ s$.

For example, the postcondition $\lambda s.\, \mu_s(s) =$ ~~own~~ of unlock in Section 2 is stable, because other-stepping cannot change the *self*-component μ_s. On the other hand, the predicate $\lambda s.\, \pi(s)$ is not stable, as already commented in Section 2, because the value of π can be changed by a thread other-stepping by Spin.set_tr.

Definition 3.16 (Atomic action). Let V be a resource and A a type. An atomic action (or action, for short) a of type A, over resource V is relation between a value $v : A$, and V-states s and s', with the properties below. We write $a\ v\ s\ s'$ to relate the values and say that a executed in input state s, and produced output state s' and return value v. The properties of a are:

(1) *(internality)* for every v, the relation $a\ v$ on states is an internal transition of V
(2) *(functionality)* v is uniquely determined by s, i.e., if $a\ v_1\ s\ s_1'$ and $a\ v_2\ s\ s_2'$, then $v_1 = v_2$

In an action a, s' is also uniquely determined by s, because for each v, the transition $a\ v$ is functional (Def. 3.9.(1)).

We can now formally define the key concept that enables program reuse by lifting: morphisms.

Definition 3.17 (Morphism). Let V and W be resources. A morphism $f : V \to W$ consists of two components:

- A relation on states $s_v \in V.S$ and $s_w \in W.S$, written $(s_v, s_w) \in f$
- A function on internal transitions $f : V.\Delta_i \to W.\Delta_i$.

The components satisfy the following properties:

(1) *(W simulates V by internal steps)* if $t \in V.\Delta_i$ and $t\ s_v\ s_v'$ and $(s_v, s_w) \in f$, then there exists s_w' such that $f(t)\ s_w\ s_w'$ and $(s_v', s_w') \in f$.
(2) *(functionality)* if $(s_{v1}, s_w) \in f$ and $(s_{v2}, s_w) \in f$, then $s_{v1} = s_{v2}$.
(3) *(V simulates W by other steps)* if $s_w \xrightarrow[W]{}^{*} s_w'$ and $(s_v, s_w) \in f$, then there exists s_v' such that $s_v \xrightarrow[V]{}^{*} s_v'$ and $(s_v', s_w') \in f$.
(4) *(frame preservation)* there exists function $\phi : U_W \to U_V$ (notice the contravariance), such that: if $(s_v, s_w \rhd p) \in f$, then $s_v = s_v' \rhd \phi\ p$, and $(s_v' \lhd \phi\ p, s_w \lhd p) \in f$.
(5) *(other-fixity)* if $(s_v, s_w) \in f$ and $(s_v', s_w') \in f$ and $a_o(s_w) = a_o(s_w')$ then $a_o(s_v) = a_o(s_v')$.

Property (1) is a relatively standard statement of simulation: whenever V can make a step by some (internal) transition t to move from s_v to s_v', then W can follow. That is, W can transition from a state s_w into s_w'. Moreover, it is required that $(s_v, s_w) \in f$ and $(s_v', s_w') \in f$. The matching step of W is constructively computed by f's transition component, in order to support program lifting in rule Lift of Section 1, i.e., the on-the-fly modification of e in V to morph $f\ e$ in W.

Functionality property (2) requires that f, when viewed as a relation on states, is a partial function from W to V (note the contravariance). This property is essential for the soundness of the Lift rule. The lifting, formally defined in Appendix B, logically functions as follows: it takes a state $s_w \in S_w$, transforms it into $s_v \in S_v$ by applying the state component of f, then simulates e's transitions, by f, starting from s_v, to compute the corresponding modification to s_w. Functionality ensures that s_v is uniquely determined from s_w, as otherwise we would not know precisely in which V state to start the simulated execution of e.

Functionality may look restrictive at the moment, as the customary definitions of simulation in the literature require the state component to be a relation, not necessarily a function. However, the property is required by the specifics of our setting. In the literature, simulations are usually considered between STSs that themselves typically represent some kind of programs. For us, the STSs are part of the program's type, and we consider how the simulation affects the program, not just the type. The additional level of consideration imposes the additional property. Nevertheless, we show in Section 5 that the restriction can be lifted by a relatively simple generalization to *indexed morphism families.*

Property (3) states a simulation in the opposite direction, i.e., V simulates W, but using the reflexive-transitive closure of other-stepping. Intuitively, the property ensures that we may view the interference in W as interference in V. Thus, a morphism f actually consists of two simulations, which work in opposite directions, but whose definitions are very different. In particular, the simulation in property (3) only depends on f's state component, and, unlike the simulation in property (1), it is not given constructively by f's transition component. For example, in Section 2.4, one may see that Spin simulates CSL in the sense of property (3), because each transition in CSL is a coupling of a transition in Spin. The reason for the difference between the two simulations is that the simulation in property (3) is not used to modify programs on the fly, but merely to ensure the soundness of the Lift rule. The premiss of Lift specifies e only under the assumption that the interfering threads respect V. Morph $f\ e$ logically executes e, modifying its transitions by f, as described above. Thus, unless we can view interference to morph $f\ e$ in W as interference to e in V, we cannot use the specification of e to infer anything about morph $f\ e$.

Properties (4) and (5) state preservation of the subjective structure between the states of V and W. Property (4) says that whenever we frame by p in W, there is a uniquely determined frame $\phi\ p$ in V that corresponds to it. For example, in the case of the morphism f : Spin $\longrightarrow$ CSL in Section 2.4, U.CSL $= U$.Spin $\times$ U.Xfer, and ϕ is defined as the first projection, following the definition of f's state component. Property (5) requires that the *other* fields are preserved by f. When f maps s_w to s_v, then $a_o(s_w)$ only depends on $a_o(s_v)$, but not on $a_s(s_w)$ and $a_j(s_w)$.

We close the section with the definition of f-stepping (i.e., stepping under a morphism $f : V \longrightarrow W$), and its associated property of f-stability. These are similar to *other*-stepping and stability (Definitions 3.14 and 3.15), but where the latter consider interference of other threads, f-stepping considers steps that are f-images of internal transitions of V. Intuitively, f-stable predicates are preserved by programs morphed by f. For example, in the Lift rule in Section 1, the morphism $f : V \longrightarrow W$ lifts the program e, and preserves the f-stable predicate I. In Section 2.4, the predicate $I \mathrel{\widehat{=}} \lambda s.\, \sigma_s(s) = h$ used to lift lock to lock$'$ is stable under morphisms f : Spin $\longrightarrow$ CSL, because the images under f of internal transitions of Spin do not modify the *self* heap in CSL.

Definition 3.18 (f-stepping). Let $f : V \longrightarrow W$ be a morphism, and s_w, s'_w be W-states. We say that s_w steps by f to s'_w, written $s_w \underset{f}{\longrightarrow} s'_w$, if one of the following is true:

(1) there exists $t \in V.\Delta_i$ and s_v, s'_v, such that $(s_v, s_w) \in f$, $(s'_v, s'_w) \in f$, $t\ s_v\ s'_v$ and $f(t)\ s_w\ s'_w$
(2) $s_w \underset{W}{\longrightarrow} s'_w$

In other words, either s_w steps into s'_w by interference on W, or the step is an f-image of an internal transition in V. We write $\underset{f}{\longrightarrow}^*$ for reflexive-transitive closure of $\underset{f}{\longrightarrow}$.

Definition 3.19 (f-stability). Let $f : V \longrightarrow W$ be a morphism. Predicate P over W-states is *f-stable in state s* if whenever $s \underset{f}{\longrightarrow}^* s'$, then $P\ s'$. P is f-stable if it is f-stable in state s for every s for which $P\ s$. Given P, we define its *f-stabilization* P^f as $P^f\ s\ \mathrel{\widehat{=}}\ \forall s'.\, s \xrightarrow[f]{}{}^{*} s' \to P\ s'$. It is easy to see that P^f is f-stable, and that P is f-stable iff $\forall s.\, P\ s \to P^f\ s$.

3.2 Basic constructions

Definition 3.20 (Identity and composition). The *identity morphism* 1_V on a resource V consists of the following state and transition components:

- $(s, s') \in 1_V$ iff $s = s'$
- for every $t \in V.\Delta_i$, $1_V(t) = t$

Let $f : V \to W$ and $g : W \to X$ be morphism. The composition morphism $g \circ f : V \to X$ consists of the following state and transition components:

- $(s, s') \in g \circ f$ iff there exists s'' such that $(s, s'') \in f$ and $(s'', s') \in g$.
- for every $t \in V.\Delta_i$, $(g \circ f)(t) = g(f(t))$

It is easy to show that $\circ$ is associative, with 1_V (resp. 1_W) as the right (resp. left) identity.

Definition 3.21 (Resource restriction). Let V be a resource, and I a global V-inductive predicate. Restriction of V by I, denoted V/I, is a resource defined over the same PCM and type as V, and with state space, flattening, and transitions defined as follows, to make I hold constantly.

(1) $(V/I).S(s)\ \mathrel{\widehat{=}}\ V.S(s) \wedge I(s)$
(2) $(V/I).\ulcorner s \urcorner\ \mathrel{\widehat{=}}\ V.\ulcorner s \urcorner$
(3) $(V/I).\Delta_i = V.\Delta_i$
(4) $t \in (V/I).\Delta_e$ if there exists $t' \in V.\Delta_e$ such that $t\ s\ s'$ iff $t'\ s\ s' \wedge I\ s'$.

There is a generic morphism from V to V/I, which is identity on states and transitions.

In (1), we conjoin I as an additional property to the state space of V. We require that I is global, so that $(V/I).S$ is global too, as required by Definition 3.5. Conditions (2-3) propagate the flattening function and internal transitions from V. Because I is inductive, the internal transitions preserve $(V/I).S$, as required by Definition 3.12. Finally, Condition (4) strengthens the external transitions of V; it requires that in V/I, an external transition can only be taken if it preserves I. The frequent use of restriction is to rule out undesired states from resource composition. We will illustrate this in Section 4, where the functionality of readers and writers is composed into a resource for readers/writers lock. Because there is a dependence between the individual resources for readers and for writers, restriction will be used to remove some state pairs from the composition.

3.3 Inference rules

The inference rules of FCSL differentiate between two different program types: ST $V\ A$ and $[\Gamma].\ \{P\}\ A\ \{Q\}@V$. The first type encompasses programs that respect the transitions of the resource V, and return a value of type A if they terminate. The second type is a subset of ST $V\ A$, selecting only those programs that satisfy the precondition P and postcondition Q. Here, Γ is a context of specification-only variables that serve to relate pre- and post-states, as illustrated in Section 2. P and Q are predicates drawn from the Calculus of Inductive Constructions (CiC) which is the logic of Coq, and A is a type in CiC.

The key concept in the inference rules is a predicate transformer vrf $e\ Q$, which takes a program e : ST $V\ A$, and postcondition Q, and returns the set of V-states from which e is safe to run, and produces an ending state and result result satisfying Q (thus, technically, $Q : A \to V\text{-state} \to \text{prop}$).

Vrf is used to encode via Hoare triple types that e has a precondition P and postcondition Q.[6]

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

In Appendix B, we define the denotational semantics in CiC for ST V A, and define the vrf predicate transformer. Thus, we can use Coq as our environment logic, and combine the Hoare triple types with other type constructors, to form higher-order computations. Here we just mention that we can now immediately give the following type to the fixed-point combinator, where T is the dependent type $T = \Pi_{x:A}.\,[\Gamma].\,\{P\}\;B\;\{Q\}@V$ of functions of argument $x : A$ producing concurrent computation with precondition P and postcondition Q:

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

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 the actual reasoning about programs, we keep the predicate transformer vrf abstract, and only rely on the following minimal set of lemmas, all proved in Coq, and presented here in separation logic notation to implicitly abstract over the current state. These, together with the typing for fix above, are the only Hoare-related rules of FCSL, though, of course, FCSL also inherits all the inference rules of CiC.

$$\begin{array}{rcl}
\mathsf{vrf_vs} & : & \mathsf{vrf}\;e\;Q \to V.S \\
\mathsf{vrf_post} & : & (\forall r\,s.\, V.S\;s \to Q_1\;r\;s \to Q_2\;r\;s) \to \mathsf{vrf}\;e\;Q_1 \to \mathsf{vrf}\;e\;Q_2 \\
\mathsf{vrf_ret} & : & V.S \to (Q\;r)^{\bullet} \to \mathsf{vrf}\;(\mathsf{ret}\;r)\;Q \\
\mathsf{vrf_bnd} & : & \mathsf{vrf}\;e_1\;(\lambda x.\,\mathsf{vrf}\;(e_2\;x)\;Q) \to \mathsf{vrf}\;(x \leftarrow e_1; (e_2\;x))\;Q \\
\mathsf{vrf_par} & : & (\mathsf{vrf}\;e_1\;Q_1) * (\mathsf{vrf}\;e_2\;Q_2) \to \mathsf{vrf}\;(e_1 \parallel e_2)\;(\lambda r : A_1 \times A_2.\,(Q_1\;r.1) * (Q_2\;r.2)) \\
\mathsf{vrf_frame} & : & (\mathsf{vrf}\;e\;Q_1) * Q_2^{\bullet} \to \mathsf{vrf}\;e\;(\lambda r.\,(Q_1\;r) * Q_2) \\
\mathsf{vrf_atm} & : & V.S \to (\lambda s.\,\exists r\,s'.\,a\;r\;s\;s' \wedge (Q\;r)^{\bullet}\;s')^{\bullet} \to \mathsf{vrf}\;(\mathsf{atomic}\;a)\;Q \\
\mathsf{vrf_morph} & : & f\hat{}(\mathsf{vrf}\;e\;Q) \wedge I^f \to \mathsf{vrf}\;(\mathsf{morph}\;f\;e)\;(\lambda r.\,f\hat{}(Q\;r) \wedge I)
\end{array}$$

The vrf_vs lemma says that if a state is in vrf e Q, then it is also in V's state space. In other words, the predicate transformer vrf is only concerned with states that are valid for the resource V.

The vrf_post lemma says that we can weaken the postcondition Q_1 into Q_2 if the first implies the second for every return value r and state s. The lemma is thus a variant of the customary Hoare logic rule of consequence. When proving Q_2 out of Q_1, it is sound to further assume $V.S$, because vrf is only concerned with states that are valid for the resource V.

The vrf_ret lemma states that if Q r holds in the initial states, then the ending state of ret r satisfies Q r; in other words, ret r does not change the state and just returns r. To account for the possibility that the environment threads may change the state, we stabilize Q r in the premiss.

The vrf_bnd lemma is the customary Dijkstra-style rule for computing a predicate transformer of a sequential composition, by nesting two applications of the transformer.

The vrf_par lemma encodes the usual property of separation logics that if the initial state s can be split into s_1 and s_2, such that e_1 executes in s_1 to obtain postcondition Q_1, and e_2 executes in s_2 to obtain postcondition Q_2, then the ending state of $e_1 \parallel e_2$ can be split in the same way. This follows from the definition of $P * Q$ which is slightly different than in separation logic, to account for FCSL's different notion of state.

$$(P * Q)\;s \;\widehat{=}\; \exists x_1\;x_2.\, a_s(s) = x_1 \bullet x_2 \wedge P\;(x_1, a_j(s), a_o(s) \bullet x_2) \wedge Q\;(x_2, a_j(s), a_o(s) \bullet x_1).$$

The definition captures the state view of the children threads e_1 and e_2 upon their forking in the parent state s. The *self*-components of the children states divide the *self*-component of the parent

[6]We abstract current state as customary in separation logic. Otherwise, the definition reads $\forall\Gamma.\; V.S\;s \to P\;s \to \mathsf{vrf}\;e\;Q\;s$.

$(a_s(s) = x_1 \bullet x_2)$. At the same time, the *other*-component of e_1 adds the *self*-components of e_2 $(a_o(s) \bullet x_2)$ to capture the fact that e_2 becomes part of the concurrent environment of e_1, and vice versa. The joint component $a_j(s)$ represents shared state, so it is propagated to both children without changing. Finally, the end-result of $e_1 \parallel e_2$ is a pair $r = (r.1, r.2)$ of type $A_1 \times A_2$, combining the return results of e_1 and e_2, of types A_1 and A_2, respectively. Thus, the postcondition of $e_1 \parallel e_2$ splits r and passes the projections to Q_1 and Q_2.

The vrf_frame lemma is, intuitively, a form of vrf_par lemma where e_2 is taken to be an idle thread. Thus, it can be seen as a combination of vrf_par and vrf_ret lemmas, which is why we stabilize Q_2 in the premiss.

The vrf_atm lemma says Q is a postcondition for an action a in the pre-state s, if there exist the return value r and post-state s' that are related by a (i.e., such that $a\ r\ s\ s'$) and $Q\ r\ s'$. We allow for environment steps before s and after s', which is why we stabilize the whole predicate binding s, and we stabilize $Q\ r$ before applying it to s'.

Finally, vrf_morph is a predicate-transformer version of Lift rule from Section 1.[7] Unfolding the definition of $f\hat{}P = \lambda s_w.\, \exists s_v.\, (s_v, s_w) \in f \wedge P\ s_v$, the lemma says that if we are given the initial W-state s_w, for which there exists s_v such that $(s_v, s_w) \in f$, and if running e in s_v results in the postcondition Q, then running morph $f\ e$ in s_w will first switch to s_v, execute e there, and then come back to obtain the ending state satisfying $f\hat{}Q$. The predicate I is propagated from the premiss to the conclusion, but is stabilized in the pre-state to avoid the side-condition that I is f-stable.

4 READERS/WRITERS

This section illustrates component reuse on the example of readers-writers locks [3, 6], a significantly more involved construction than CSL from Section 2. The writers lock wr protects a shared heap, just as in the case of CSL. When a writer acquires wr, it gains exclusive ownership of the heap. But when a reader acquires wr, the heap becomes shared by all concurrent readers, while becoming inaccessible to writers. To support this discipline, the readers have to register (resp. deregister) themselves, by incrementing (resp. decrementing) a shared counter ct that keeps the overall number of readers. The counter ct is protected by another lock rd, as shown by the prologue (resp. epilogue) procedure below.

```
prologue() =                    epilogue() =
  lock(rd);                       lock(rd);
  x ← !ct;                        x ← !ct;
  if x = 0 then lock(wr);         ct := x − 1;
  ct := x + 1;                    if x = 1 then unlock(wr);
  unlock(rd)                      unlock(rd)
```

The first reader to execute prologue is responsible for acquiring wr, and the last reader to execute epilogue releases it, to let the writers in. Moreover, epilogue should only be invoked by a reader that already went through prologue. Between calls to prologue and epilogue, the reader can freely read from the shared heap, which is guaranteed not to be changed by a writer. A thread may invoke prologue and register as a reader multiple times. The extra registrations are not extraneous as, upon forking, they are divided between the thread's children. Thus, a thread holding more than one registration is simply pre-registering its children as readers.

From the logical standpoint, prologue and epilogue manage the ownership of the protected heap, just as CSL did, but here the ownership discipline is much more involved. Intuitively, we have two

[7] Indeed, the latter is a direct consequence of vrf_morph and the definition of Hoare triple type.

	RWLock					
	WLock			RLock		
	Spin′(wr)	Shar	Priv	Spin′$_2$(rd)	Count	Shar$_2$
	$\mu_s, \pi, \lambda, \mu_o$	σ_j, ν	σ_s, σ_o	$\mu_{s2}, \pi_2, \lambda_2, \mu_{o2}$	$\kappa_s, \iota, \kappa_o$	σ_{j2}, ν_2
wrlock_tr	lock_tr(own)	–	–	–	–	–
wrunlock_tr	unlock_tr(own)	–	–	–	–	–
freeze_tr	lock_tr(~~own~~)	–	–	id_tr($\lambda s.\ \mu_{s2}(s) =$ own)	–	–
unfreeze_tr	unlock_tr(~~own~~)	–	–	id_tr($\lambda s.\ \mu_{s2}(s) =$ own)	–	–
rdlock_tr	–	–	–	lock_tr(own)	–	–
rdunlock_tr	–	–	–	unlock_tr(own)	–	–
incr_tr	–	–	–	id_tr($\lambda s.\ \mu_{s2}(s) =$ own)	incr_tr	–
decr_tr	–	–	–	id_tr($\lambda s.\ \mu_{s2}(s) =$ own)	decr_tr	–
open_tr	set_tr(own)(false)	give_tr	take_tr	–	–	–
close_tr	set_tr(own)(true)	take_tr	give_tr	–	–	–
toreader_tr	set_tr(~~own~~)(false)	give_tr	–	id_tr($\lambda s.\ \mu_{s2}(s) =$ own)	set_tr(true)	take_tr
towriter_tr	set_tr(~~own~~)(true)	take_tr	–	id_tr($\lambda s.\ \mu_{s2}(s) =$ own)	set_tr(false)	give_tr

Fig. 1. Coupling of the transitions of RWLock. The rows are the transitions of RWLock, and the columns are the transitions of individual components which are coupled to provide the RWLock transition. The top row of each column lists the fields of the component's state space. Empty cells indicate the id_tr transition. All the transitions of RWLock are internal.

distinct resources: WLock for writers, and RLock for readers. When the heap is in the shared state of WLock, it can be acquired by a writer and moved to the writer's private state. We say that the heap is then in "write" mode. This is the functionality we already saw in CSL. But here, the heap can also be acquired by the first reader that goes through prologue, in which case the heap moves to the shared state of RLock, where it can be accessed by any reader. We say that the heap is in "read" mode. Dually, epilogue returns the heap from the shared state of RLock to the shared state of WLock, when invoked by the last reader.

We can thus divide the readers/writers construction into several sub-components. First, we formalize the two different ownership modes by a new resource Spin′. Spin′ will implement the "write" mode, similar to Spin in Section 2, but will also enable the "read" mode to be added by composition with other resources. Second, we formalize the discipline of reader registration and deregistration, and ensure that the protected heap is in "read" mode if a registered reader exists. Finally, we formalize the transfer of the protected heap between different ownership modes, by composing instances of the resources Shar and Priv that we already introduced in Section 2. Ultimately, the pieces combine into the resource RWLock for readers/writers, as schematically illustrated in Figure 1. We will explain the figure in detail further in this section; for now, it suffices to note that the construction instantiates each of Spin′ and Shar twice (once for writers, once for readers), thus achieving reuse.

Our description of RWLock will focus on the prologue and epilogue procedures, to which we ascribe the following specifications. [8]

$$\text{prologue} : [h, c].\ \{\lambda s.\ \sigma_s(s) = h \wedge \kappa_s(s) = c\}\ \{\lambda s.\ \sigma_s(s) = h \wedge \kappa_s(s) = c + 1\} @ \text{RWLock}$$
$$\text{epilogue} : [h, c].\ \{\lambda s.\ \sigma_s(s) = h \wedge \kappa_s(s) = c + 1\}\ \{\lambda s.\ \sigma_s(s) = h \wedge \kappa_s(s) = c\} @ \text{RWLock}$$

[8]The specifications can be simplified by taking h = empty and $c = 0$; the general case can be recovered by framing.

In the specifications, σ_s stands for the private heap of the invoking thread, and κ_s is the number of readers that the thread has registered. The registration count is increased by prologue and decreased by epilogue. A thread is a reader if its $\kappa_s(s) > 0$. Notice that $\kappa_s(s)$ is a *self*-field, which has two important consequences. First, as described in Section 3, the thread's value of $\kappa_s(s)$ is divided upon forking between the thread's children, which thereby inherit any extra registrations that the parent may have had. Second, if a thread is a reader, i.e., $\kappa_s(s) > 0$, then it remains so under interference, as $\kappa_s(s)$ cannot be changed by other threads. A thread can stop being a reader only if it deregisters itself by invoking epilogue.

4.1 The resource $\mathsf{Spin}'(r)$ for locking r without exclusive ownership of r

The $\mathsf{Spin}'(r)$ resource implements spin locks, but with two different modes of ownership: exclusive ownership by the locking thread, and non-exclusive ownership. In the instance $\mathsf{Spin}'(wr)$ used by WLock, exclusive ownership is used when the writer takes the writer lock (the "write" mode of the heap), and non-exclusive ownership is used when the reader takes the writer lock (the "read" mode): in the latter case, the heap collectively must be owned by all the readers. In the instance $\mathsf{Spin}'(rd)$ used by RLock, exclusive ownership is used when the reader takes the reader lock, while non-exclusive ownership is not needed.

Omitting r from now on, the states of Spin' have the form $s = (\mu_s, (\lambda, \pi), \mu_o)$. The boolean λ is true if the underlying lock is taken, and is false otherwise. As in Spin, π is a boolean that has to be set before unlocking; $\mu_s, \mu_o \in O$ indicate the exclusive ownership of the lock; and $\mu(s) = \mu_s(s) \bullet \mu_o(s)$.

$$\begin{aligned} S(s) &\;\widehat{=}\; \mathsf{defined}\,(\mu(s)) \wedge r \neq \mathsf{null} \wedge (\neg\lambda(s) \to \mu(s) = \overline{\mathsf{own}} \wedge \pi(s)) \\ \ulcorner s \urcorner &\;\widehat{=}\; r \Mapsto \lambda(s) \end{aligned}$$

The state space imposes the condition that if the (readers or writers) lock is free ($\neg\lambda(s)$), then no thread owns the lock exclusively ($\mu(s) = \overline{\mathsf{own}}$). However, *it does not* impose the implication in the other direction: it may be that the lock is taken *and* $\mu(s) = \overline{\mathsf{own}}$, which models the non-exclusive ownership. Additionally, if the lock is free, then $\pi(s)$; that is, the shared heap will satisfy the invariant in the eventual composition with a resource for heap transfer, just like in Spin.

The transitions are similar to Spin, except they now use $\lambda(s)$ to express the lock's status, and they have to deal with two different ownership modes. We capture the latter by adding an extra parameter $x \in O$ to all non-idle transitions. Passing $x = \mathsf{own}$ (resp. $x = \overline{\mathsf{own}}$) gives us the transition dealing with exclusive (resp. non-exclusive) ownership.

$$\begin{aligned} \mathsf{lock\_tr}\; x\;\; s\; s' &\;\widehat{=}\; \neg\lambda(s) \wedge \lambda(s') \wedge \mu_s(s') = x \wedge \pi(s') \\ \mathsf{unlock\_tr}\; x\; s\; s' &\;\widehat{=}\; \lambda(s) \wedge \mu_s(s) = x \wedge \mu_o(s) = \overline{\mathsf{own}} \wedge \pi(s) \wedge \neg\lambda(s') \\ \mathsf{set\_tr}\; x\; b\; s\; s' &\;\widehat{=}\; \lambda(s) \wedge \mu_s(s) = x \wedge \mu_o(s) = \overline{\mathsf{own}} \wedge \lambda(s') \wedge \mu_s(s') = x \wedge \pi(s') = b \end{aligned}$$

For example, lock_tr switches λ from false to true, as one would expect. As in Spin, it also sets $\pi(s')$. But, if invoked with $x = \mathsf{own}$, it also sets $\mu_s(s')$ to own to signal the exclusive ownership of the lock. Similarly, unlock_tr switches λ from true to false, and also requires $\pi(s)$ to be set, as in Spin. If invoked with $x = \mathsf{own}$ it requires that the invoking thread actually has exclusive ownership of the lock. Otherwise, if invoked with $x = \overline{\mathsf{own}}$, no thread is allowed to have exclusive ownership ($\mu_s(s) = \mu_o(s) = \overline{\mathsf{own}}$). The transitions obtained for different values of x will be coupled differently in the eventual composition. Importantly, the $x = \mathsf{own}$ versions of the transitions are *internal*, whereas those obtained with $x = \overline{\mathsf{own}}$ are *external*, as the notion of ownership that the latter represents will be formalized only when we compose with the resource for readers. The set_tr x b transition sets $\pi(s')$ to b. It requires the lock to be held ($\lambda(s)$), but not exclusively by other threads ($\mu_o(s) = \overline{\mathsf{own}}$). Thus, in the composition, π could be changed by any reader, if the readers have acquired the writer lock, but only by the writer that owns the lock. It may be interesting to observe here that passing $x = \mathsf{own}$ to the transitions essentially recovers the functionality of Spin, whereas passing $x = \overline{\mathsf{own}}$ produces new transitions. If we strengthen the state space of Spin′ to include $\lambda(s) \longrightarrow \mu(s) = \mathsf{own}$, then none of the new transitions can ever be invoked, because the conditions on their initial state will never be satisfiable. Thus, Spin′ reduces to Spin, when $x = \mathsf{own}$.

4.2 The counting resource Count

The resource Count tracks reader registration. Physically, the registration count is kept in the pointer ct, but it is the division of the count into *self* and *other* fields that is important for the specification of prologue and epilogue. The states of Count thus have the form $s = (\kappa_s, \iota, \kappa_o)$, where κ_s and κ_o keep the number of registrations made by the invoking thread and its environment, respectively. In every resource, the *self* and *other* components must be drawn from the PCM; here it is the PCM of natural numbers under +, with 0 as the unit element. The field ι is a boolean, motivated similarly to π in Section 2–it indicates in the eventual composition of Count into RLock that the heap on which the readers are to operate is in "read" mode. The above description motivates the following state-space design for Count.

$$
\begin{aligned}
S(s) &\mathrel{\widehat{=}} \mathsf{ct} \neq \mathsf{null} \wedge \kappa(s) > 0 \longrightarrow \iota(s) \\
\ulcorner s \urcorner &\mathrel{\widehat{=}} \mathsf{ct} \Mapsto \kappa(s)
\end{aligned}
$$

The conjunct $\kappa(s) > 0 \longrightarrow \iota(s)$ ensures that if there are registered readers, then, in the composition, the heap is in "read" mode. The conjunct $\mathsf{ct} \neq \mathsf{null}$ requires that ct is a valid pointer.

The non-idle transitions of Count are as follows.

$$
\begin{aligned}
\mathsf{incr\_tr}\ s\ s' &\mathrel{\widehat{=}} \iota(s) \wedge \kappa_s(s') = \kappa_s(s) + 1 \\
\mathsf{decr\_tr}\ s\ s' &\mathrel{\widehat{=}} \kappa_s(s') + 1 = \kappa_s(s) \wedge \iota(s') \\
\mathsf{set\_tr}\ b\ s\ s' &\mathrel{\widehat{=}} \kappa(s) = \kappa(s') = 0 \wedge \iota(s') = b
\end{aligned}
$$

In English, incr_tr increments $\kappa_s(s)$, but requires that the $\iota(s)$ bit is set, that is, the heap is in "read" mode. Similarly, decr_tr decrements $\kappa_s(s)$, but the latter has to be non-zero—a reader can cancel only the registration that it had made itself. By the definition of S, if $\kappa_s(s) > 0$ in the pre-state, then $\iota(s)$ is set, and decr_tr keeps ι set in the post-state. If $\kappa_s(s) = 0$, then decr_tr cannot execute. Set_tr b sets $\iota(s')$ to b, but it requires (and maintains) that $\kappa(s) = 0$; that is, the ownership mode of the heap can be changed only when there are no readers in the system.

4.3 Composing into RWLock

We now combine the components into a resource RWLock, as shown in Figure 1. The fields of the combination contain the fields of WLock, tracking information about writers, and of RLock, tracking information about readers. The WLock state is itself a product of the state-spaces of Spin′(wr), Shar, and Priv. Here, Shar provides the functionality of a shared heap with an invariant I. When the protected heap is in this sub-resource, it is in WLock, but is not owned by any thread. Priv provides the functionality of private heaps, with the operations for lookup, update, allocation and deallocation, whose discussion we elide here. When the heap is in Priv, it is owned exclusively by a writer that locked it, i.e., the heap is in the "write" mode. The RLock state is a product of the state-spaces of Spin′(rd), Count and Shar. Here Spin′(rd) provides the functionality of the spin lock rd. Shar provides the functionality of a shared heap with an invariant I. When the protected heap is in this sub-resource, it is in RLock, and owned collectively by all readers, that is, it is in "read" mode. To differentiate these instances of Spin′ and Shar from the ones used in WLock, we index them and their fields by 2.

The state space of RWLock, however, cannot be a simple product of the underlying components, and we need to impose the additional invariant RWinv defined below. Thus, we first build an intermediate resource RWLock′ which combines the states and transitions as shown in Figure 1, then construct the restriction RWLock = RWLock′/RWinv (see Definition 3.21), and inject RWLock′ into RWLock by the generic morphism for resource restrictions.

$$\mathsf{RWinv}(s) \quad \widehat{=} \quad \pi(s) = \nu(s) \land \iota(s) = \nu_2(s) \land \pi_2(s) \land (\lambda_2(s) \to \mu_{s2}(s) = \mathsf{own}) \land \\ (\nu_2(s) \leftrightarrow \lambda(s) \land \mu(s) = \cancel{\mathsf{own}} \land \neg\nu(s))$$

The first and second conjuncts of RWinv capture that π in Spin′(wr) and ι in Count are proxies for the presence of the protected heap in Shar and Shar_2, respectively. This is similar to how we equated π and ν in the state space of CSL in Section 2. The third conjunct fixes the value of $\pi_2(s)$, indicating that we are not going to be coupling unlock_tr of Spin′(rd) in non-trivial ways. The fourth conjunct excludes the possibility for the collective ownership of rd, as the reader lock will always be acquired exclusively by readers. Finally, the last conjunct describes the possible states in which the protected heap may be. It says that the protected heap is in RLock ($\nu_2(s)$) iff the writer lock is taken ($\lambda(s)$) by readers ($\mu(s) = \cancel{\mathsf{own}}$), and the heap is not in WLock ($\neg\nu(s)$).

The wrlock_tr and wrunlock_tr in Figure 1 are transitions for exclusive locking and unlocking by the writer. Thus, they lift the locking and unlocking transition from Spin′(wr), and do so by coupling with identity transitions across the board. We use the own version of the transition, i.e., locking and unlocking for exclusive ownership. The transitions freeze_tr and unfreeze_tr correspond to the reader locking and unlocking the writer lock, respectively, and thus couple with the ~~own~~ version of Spin′(wr) locking and unlocking transitions. They also require in Spin′(rd) that the reader lock is owned. Hence, a reader can try to lock and unlock the writers lock, but only if she first obtains the readers lock. We emphasize how the relationship between the various fields ensures that freeze_tr and unfreeze_tr can only be invoked when $\kappa(s) = 0$, i.e., the invoking reader is the sole reader in the system, and has not yet incremented $\kappa(s)$ (first reader), or has just decremented $\kappa(s)$ (last reader). Indeed, if $\kappa(s) > 0$ then $\iota(s)$ by Count.S. But then, $\nu_2(s)$ by RWinv, and then also $\lambda(s)$, $\neg\nu(s)$ and $\neg\pi(s)$. But the subcomponent lock_tr(~~own~~) of freeze_tr requires $\neg\lambda(s)$, and the subcomponent unlock_tr(~~own~~) of unfreeze_tr requires $\pi(s)$.

The transitions rdlock_tr and rdunlock_tr implement the locking and unlocking of the readers lock, and thus invoke the respective own version of the Spin′(rd) transitions. The incr_tr and decr_tr are straightforward lifting from Count, but can only be invoked in the combination by a thread holding the reader lock.

Finally, the last four transitions implement the ownership transfer of the heap within a WLock resource, and between WLock and RLock. Transitions open_tr and close_tr move the heap between WLock shared state (when the heap is not owned by anybody) and the writers resource private heap ("write" mode). On the other hand, toreader_tr moves the heap from WLock to RLock, setting the heap to "read-only" mode. Notice how the transition synchronizes the boolean fields π in Spin′(wr) and ι in Count, to capture that the first is set to false simultaneously with the second being set to true. Transition towriter_tr works in the opposite direction.

4.4 Annotating and verifying prologue

We next present the proof outline for prologue in Figure 2 (the similar proof for epilogue is in the Coq files). In the code, we replace the physical operations such as, e.g., reading from ct and writing into it, with actions. Actions thus decorate the physical operations with auxiliary code, built out of the transition of RWLock, and the program erases to the one given in Section 4.

In line 2, rdlock is a procedure that loops over the spin-lock rd, trying to acquire it by means of rdlock_tr transition in RWLock. The latter is a coupling of $\mathsf{Spin}'_2(rd)$.lock_tr(own) with id_tr prologue() =

1. $\{\sigma_s(s) = h \wedge \kappa_s(s) = c\}$
2. rdlock;
3. $\{\sigma_s(s) = h \wedge \kappa_s(s) = c \wedge \mu_{s2}(s) = \mathsf{own}\}$
4. $x \leftarrow$ atomic (readcnt_act);
5. $\{\sigma_s(s) = h \wedge \kappa_s(s) = c \wedge \mu_{s2}(s) = \mathsf{own} \wedge x = c + \kappa_o(s)\}$
6. if $x = 0$ then freeze; atomic (toreader_act);
7. $\{\sigma_s(s) = h \wedge \kappa_s(s) = c \wedge \mu_{s2}(s) = \mathsf{own} \wedge x = c + \kappa_o(s) \wedge \iota_s(s)\}$
8. atomic (incr_act x);
9. $\{\sigma_s(s) = h \wedge \kappa_s(s) = c + 1 \wedge \mu_{s2}(s) = \mathsf{own}\}$
10. atomic (rdunlock_act)
11. $\{\sigma_s(s) = h \wedge \kappa_s(s) = c + 1\}$

Fig. 2. Proof outline for prologue.

on all sub-components (Figure 1). Thus, it sets μ_{s2} to own, preserving the other components. In particular, the values of σ_s and κ_s are propagated from line 1 to line 3. For brevity, we omit the definition of rdlock; it is implemented by lifting, and thus reusing, the lock procedure for Spin′, exactly in the same way that we produced lock′ out of lock in Section 2.

The action readcnt_act is defined as follows.

$$\mathsf{readcnt\_act}\ x\ s\ s' \ \widehat{=}\ \mathsf{id\_tr}\ (\lambda s.\, \kappa(s) = x)\ s\ s'$$

As it invokes id_tr, the action does not change the state, but the predicate $\lambda s.\, \kappa(s) = x$ ties the return result x to $\kappa(s)$, which equals the contents of ct. Thus, read_act erases to a lookup of ct.

Line 6 ensures that the protected heap is acquired by the readers. If $x > 0$, then by the state space of Count, we know that $\kappa(s) > 0$ and thus, $\iota(s)$. On the other hand, if $x = 0$, we invoke freeze; toreader_act. Freeze is a locking procedure, just like rdlock. However, it loops over *wr*, trying to execute the freeze_tr transition, which is composed out of Spin.lock_tr($\overline{\mathsf{own}}$) with a number of idle transitions. In the outcome, the loop terminates with *wr* lock taken, and $\nu(s)$ field set, indicating that the protected heap is in the writer resource. Thus, we subsequently execute toreader_act to move the heap to the reader resource, and thus set $\nu_2(s)$. As the invariant RWinv equates $\nu_2(s) = \iota(s)$, we know that $\iota(s)$ holds in line 7. Thus, we can invoke incr_act x, defined as:

$$\mathsf{incr\_act}\ x\ s\ s' \ \widehat{=}\ x = \kappa(s) \wedge \mathsf{incr\_tr}\ s\ s'$$

The action transitions by incr_tr to increment $\kappa_s(s)$. It requires $\kappa(s)$, which is the contents of ct, to equal x; hence, it erases to the physical operation of writing of $x + 1$ into ct. Finally, in line 10, rdunlock_act invokes RWLock.rdunlock_tr to release the *rd* lock, giving us the final specification.

5 INDEXED MORPHISM FAMILIES AND QUIESCENCE

As defined in Section 3, the state component of a morphism $f : V \longrightarrow W$ is a (partial) function from $W.S$ to $V.S$. Functionality is required for f to be able to lift programs from V to W. Indeed, given a program e over V, and a W-state s_w, lifting requires first mapping s_w into a V-state s_v, in order to run e on s_v. It is only sensible for s_v to be uniquely determined by s_w, and we were not able to prove the Lift rule sound without functionality.

There are examples, however, as we will show, where we would like f to be a relation on states, but not a function. To reconcile the two contradictory requirements, we generalize morphisms 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 of f is a partial function $f : X \to W.S \rightharpoonup V.S$, and the transition component of f is a function $f : X \to V.\Delta \to W.\Delta$, satisfying a number of properties (listed in Appendix A), which reduce to properties of morphisms when X is the unit type. By choosing X suitably, we can represent any relation $R \subseteq W.S \times V.S$ as a partial function $f_R : X \to W.S \rightharpoonup V.S$. Indeed, we can take $X = V.S$, and set $f_R\ s_v\ s_w = s_v$ if $(s_w, s_v) \in R$, and undefined otherwise. The morph constructor, and the LIFT rule are generalized to receive the initial index x, and postulate the existence of an ending index y in the postcondition, as follows.

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

As an illustration, consider a *history-based* specification of a concurrent stack's push method [25].

$$\begin{array}{l}\mathsf{push}(v) : [\tau].\ \{\lambda s.\ \sigma_s(s) = \mathsf{empty} \wedge \tau_s(s) = \mathsf{empty} \wedge \tau \sqsubseteq \tau_o(s)\}\\ \qquad\qquad \{\lambda s.\ \sigma_s(s) = \mathsf{empty} \wedge \exists t\ vs.\ \tau_s(s) = t \Mapsto (vs, v :: vs) \wedge \forall t' \in \mathsf{dom}(\tau).\ t' < t\}@\mathsf{Stack}\end{array}$$

The Stack states have the fields $s = ((\sigma_s, \tau_s), (\sigma_j, \alpha), (\sigma_o, \tau_o))$, where $\sigma_s, \sigma_j, \sigma_o \in$ heap and $\tau_s, \tau_o \in$ hist. The heaps σ_s, σ_o are used to allocate new cells before pushing them onto the stack. The heap σ_j stores the stack's physical layout, and α is the abstract contents of the stack. The full definition of Stack.S is not important for the discussion here; it suffices to know that we have a predicate layout such that $\forall s \in \mathsf{Stack}.S.\ \mathsf{layout}\,\alpha(s)\,\sigma_j(s)$, i.e., layout describes how α is laid out in σ_j. Histories τ_s and τ_o are finite maps sending a time-stamp t to an abstract description of an operation performed at time t. For example, the singleton history $42 \Mapsto (vs, v :: vs)$, denotes that at time 42, the element v was pushed onto the stack, thus changing α from the sequence vs to $v :: vs$. Histories are a PCM under the operation of disjoint union (undefined if operands share a time-stamp), and with the empty history as unit. If $t \in \mathsf{dom}(\tau_s)$ (resp. $t \in \mathsf{dom}(\tau_o)$), then the operation at time t was executed by the specified thread (resp. the environment). For example, push starts with $\tau_s(s) = \mathsf{empty}$ and ends with $\tau_s(s) = t \Mapsto (vs, v :: vs)$ to indicate that $\mathsf{push}(v)$ indeed pushed v. The interfering threads may have executed their own operations before and after t, to change the value of τ_o. The conjunct $\forall t' \in \mathsf{dom}(\tau).\ t' < t$ temporally orders t after the timestamps of all the operations that terminated before $\mathsf{push}(v)$ was invoked.

Now consider the program $e = \mathsf{push}(1) \parallel \mathsf{push}(2)$, whose type derivation is in the Coq files.

$$\begin{array}{l}e : \{\lambda s.\ \sigma_s(s) = \mathsf{empty} \wedge \tau_s(s) = \mathsf{empty}\}\\ \quad \{\lambda s.\ \sigma_s(s) = \mathsf{empty} \wedge \exists t_1\ vs_1\ t_2\ vs_2.\ \tau_s(s) = t_1 \Mapsto (vs_1, 1 :: vs_1) \bullet t_2 \Mapsto (vs_2, 2 :: vs_2)\}@\mathsf{Stack}.\end{array}$$

The specification reflects that e pushes 1 and 2, to change the stack contents from vs_1 to $1 :: vs_1$ at time t_1, and from vs_2 to $2 :: vs_2$ at time t_2. The order of pushes is unspecified, so we do not 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 postcondition does not imply an ordering between t_1 and t_2). Moreover, we do not 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 cannot 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 *quiescently*, i.e., without interfering threads? For example, a program working over the resource Priv from Section 2 (hence, containing only σ_s and σ_o), can invoke e over the empty stack installed in σ_s. 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 $[1, 2]$, or $[2, 1]$? The latter can even be stated without histories, using solely heaps in Priv, as follows.

$$\{\lambda s.\ \mathsf{layout\ nil}\ \sigma_s(s)\}\ \{\lambda s.\ \mathsf{layout}\,[1, 2]\,\sigma_s(s) \vee \mathsf{layout}\,[2, 1]\,\sigma_s(s)\}@\mathsf{Priv}$$

We would thus like a morphism $f : \mathsf{Stack} \to \mathsf{Priv}$ that "erases histories", but such a morphism cannot be constructed. Its state component should map a Priv-state, containing only heaps, to a

Stack-state, containing histories as well, and thus has to "invent" the history component out of thin air. This is where families come in. We make $f : \mathsf{Stack} \xrightarrow{\mathsf{hist}} \mathsf{Priv}$ a family over $X = \mathsf{hist}$, thereby passing to f the history τ that should be added to a Priv state in order to produce a Stack state. We define f's state component as follows, where we use the notation $(s_w, s_v) \in f\ \tau$ instead of $f\ \tau\ s_w = s_v$, to emphasize the partiality of f.

$$(s_{\mathsf{Priv}}, s_{\mathsf{Stack}}) \in f\ \tau \quad \widehat{=} \quad \sigma_s(s_{\mathsf{Priv}}) = \sigma_s(s_{\mathsf{Stack}}) \bullet \sigma_j(s_{\mathsf{Stack}}) \wedge \sigma_o(s_{\mathsf{Priv}}) = \sigma_o(s_{\mathsf{Stack}}) \wedge \tau_s(s_{\mathsf{Stack}}) = \tau \wedge \tau_o(s_{\mathsf{Stack}}) = \mathsf{empty}$$

The first conjunct directly states that Stack is installed in $\sigma_s(s_{\mathsf{Priv}})$ by making one chunk of $\sigma_s(s_{\mathsf{Priv}})$ be the joint heap $\sigma_j(s_{\mathsf{Stack}})$, and the other chunk be $\sigma_s(s_{\mathsf{Stack}})$.[9] The second conjunct says that the heap $\sigma_o(s_{\mathsf{Priv}})$ of the interfering threads is propagated to $\sigma_o(s_{\mathsf{Stack}})$. The third conjunct captures that the history component of s_{Stack} is set to the index τ, as discussed immediately above. Finally, in the last conjunct, the $\tau_o(\mathsf{Stack})$ history is declared empty, thus directly formalizing quiescence. We elide the definition of f's transition component, because we also elided the definition of Stack.[10] Now, applying the LiftX rule to the Stack specification of e, with $I\ x$ being the always-true predicate on Priv states, and $x = \mathsf{empty}$, gives us exactly the desired Priv specification, after some trivial rearrangements.

6 RELATED WORK

There have been several approaches to relating concurrent resources, including simultaneous modifications to their states, and program lifting.

Higher-order auxiliary code. One approach, originated by Jacobs and Piessens [14], and later expanded by Svendsen et al. [27, 28], relies on parametrizing a program and its proof with auxiliary code that works over the state of other resources. For example, using the names from Sections 1 and 2, a locking program over Spin can be parametrized by an auxiliary function over Xfer which, once executed, transfers the shared heap in Xfer to private state, much like the transition Xfer.open_tr would in Section 2. The locking program should be implemented so as to invoke this auxiliary function at the moment of successful locking. In contrast, we formalized the scenario in Section 2 by exhibiting a morphism from Spin to the extended resource CSL that couples Spin with Xfer. Once Spin locks, the heap transfer in CSL does not occur automatically, but the CSL resource is placed in a state where the transfer can be executed by invoking open_tr. This is somewhat less immediate than parametrization, but sufficient for our main goal, which is reusing Spin's implementation of locking without reverification. One advantage of our approach is that lifting a program from the source to the target resource is done after the program has been implemented, and only depends on the program's type (i.e., the pre/postcondition, and the definition of the two resources), whereas with parametrization, the program has to be developed with the parameter auxiliary functions in mind from the very beginning. A well-known challenge of parametrizing a program by an auxiliary function is exhibited when the point at which to execute the auxiliary

[9] As we want to build s_{Stack} out of s_{Priv}, we have to identify a chunk of $\sigma_s(s_{\mathsf{Priv}})$, which we want to assign to $\sigma_j(s_{\mathsf{Priv}})$. Moreover, this chunk has to be unique, else f will not satisfy the functionality property (2) of Definition 3.17. We ensure uniqueness by insisting that the predicate layout is precise – a property commonly required in separation logics.

[10] In our Coq files, we carried out the development for a Treiber variant of concurrent stacks, with some minor Treiber-specific modifications. We have also applied a similar morphism to a program constructing a spanning tree of a graph in place by marking and pruning the graphs' edges. There, the morphism was essential for showing that the tree constructed by pruning is spanning, i.e., it contains all the graph's nodes.

function can be determined only after the program has already terminated. We *expect* our morphisms to scale to such cases, precisely because lifting depends only on the program's type, not the code (hence, termination is irrelevant). However, this remains to be confirmed.

Abstract atomicity. Another approach, originated by Da Rocha-Pinto et al. [7] in TaDA logic, and recently adopted by Iris [16], introduces a new judgment form, $\langle P \rangle\ e\ \langle Q \rangle$, capturing that e has a precondition P and postcondition Q, but is also *abstractly atomic* in the following sense: e and its concurrent environment maintain the validity of P through the execution, until at one point e makes an atomic step that makes Q hold. After that point, Q may be invalidated, either by future steps of e, or by the environment. The challenge of this approach is that the new judgment has a rather complicated proof theory, and comes with auxiliary concepts, such as atomicity tokens, that impose some restrictions. For example, programs with helping, where one thread executes the work on behalf of another, currently are not supported by TaDA because their verification requires atomicity tokens to exchange ownership. In contrast, for us, ownership transfer is encoded by transition coupling, and is thus directly addressed by morphisms and simulations. We have been able to easily support helping, and have verified, in our Coq files, the flat combiner algorithm [12], a non-trivial helping example. We also verified representative clients that couple the transitions of the flat combiner with non-idle transitions of another resource. These latter transitions are to be executed simultaneously with the flat-combiner helping. The abstract atomicity approach, either in TaDA or Iris, also does not consider simulation as a way of relating resources.

The Iris version of abstract atomicity differs from the one of TaDA in that it is encoded using higher-order state available in Iris's model. Otherwise, the fragment of Iris's proof theory that handles abstract atomicity is almost identical to that of TaDA. Similarly to SCSL [17], FCSL [21], and the current paper, Iris uses PCMs to encode auxiliary state. Iris also encodes STSs via PCMs, but that is a move that we resist here. The structure-preserving functions between PCMs (aka. *local actions* [5]) are significantly different from structure-preserving functions between STSs that we consider here in the form of morphisms, which is why we avoid conflating the two. Finally, while in this paper we do not consider higher-order state, we expect that our morphism-based approach should easily reconcile with it. In particular, we expect that the Lift rule could be proved sound in Iris's model (if extended with morphisms), but this is an orthogonal consideration.

Protocol hooks. Concurrently with us, Sergey et al. [11] 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 associated with the hook is satisfied. In this sense, hooks implement a form of our transition coupling, but where one operand is the idle transition id_tr P, with P the associated condition. The above version of Disel does not consider transition coupling where both operands are non-idle (which we needed in Figure 1 to define, for example, the toreader_tr transition, and in the flat combiner implementation in our Coq files), or notions of morphism and simulation. Our work does not consider distributed protocols.

Refinement reasoning and linearizability. In a somewhat different, relational, flavor of separation logics [18, 19, 30], and more generally, in the work on proving linearizability [4, 13, 24], the approaches explicitly establish a simulation between two programs; typically one concurrent, the other sequential. This is required for showing that a concurrent program is logically atomic; that is, it linearizes to the given sequential program. Our goal in this paper is somewhat different. Instead of establishing a simulation between two programs, we establish a simulation (i.e., a morphism) between two STSs, which are components of program types, but are themselves not programs. Simulation between STSs is easier to establish than simulation between programs, as STSs have a much simpler structure—being transition systems, they omit programming constructions such as conditionals, loops, local state, or function calls. Thus, our simulation does not *directly* prove that a program is linearizable, but is intended for lifting a program from the source to the target STS, without reproving. Logical atomicity should be handled by other components of the system. For example, recent related work on FCSL [8], shows that specifications based on PCMs with *self* and *other* components can specify logical atomicity, even for sophisticated algorithms with future-dependent linearization points [15].

Previous work on FCSL. The current paper builds on the previous work on FCSL [21], to which it adds a novel notion of morphism, and significantly modifies the definition of concurrent resources. In FCSL, each concurrent resource was a finite map from labels (natural numbers) to sub-components. For example, using the concepts from Section 2, one could represent CSL as a finite map $l_1 \Mapsto \mathsf{Spin} \uplus l_2 \Mapsto \mathsf{Xfer}$, where l_1 and l_2 are labels identifying Spin and Xfer, respectively. This approach provides interesting equations on resources; for example, one can freely rearrange the finite map components by using commutativity and associativity of $\cup$. However, it also complicates mechanized verification, because one frequently needs to prove that a label is in the domain of a map, before extracting the labeled component. In the new version of FCSL, we significantly reduce the sizes of mechanized proofs by removing labels and combining components by means of pairing their states (Definitions 3.7 and 3.8 in Section 3). Consequently, if we changed the definition of CSL in Section 2 into CSL′ by commuting Spin and Xfer throughout the construction, then CSL and CSL′ would not 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. Previously, FCSL supported quiescence by means of a dedicated and complex inference rule. In Section 5, we show that quiescence reduces to LiftX rule, via indexed morphism families.

7 CONCLUSIONS AND FUTURE WORK

This paper argues that a notion of simulation to relate resources, and the corresponding notion of morphism that allows lifting programs, are key components of modular reasoning about concurrent programs. We apply these notions in FCSL, a separation logic for fine-grained concurrency. Our preliminary experiments indicate that the formalism leads to significant shortening of mechanized proofs and reuse of resource definitions and program verifications. Given a morphism from resource V to resource W, programs written over V can automatically be lifted to work over W, and the lifting is realized by means of a single Hoare-style inference rule. We call our notion of morphism “subjective simulation”, because it applies to STSs with subjective division of states into *self* and *other* components. A morphism exhibits a form of forward simulation [1] of V by W. The morphism is also interference-aware, as it exhibits a form of simulation of W by V, performed on transposed states, where the *self* and *other* components are swapped.

Morphisms are useful for a number of applications. One is lifting a program from V to W, when W includes V as a sub-component. This was illustrated in Section 4, where we built a resource for readers/writers lock in a staged, decomposed, manner. Another application is in managing the scope of auxiliary state. This was illustrated in Section 5, where auxiliary state of histories is introduced within the scope of a morphism that maps abstract stacks to their underlying heaps. Such histories should be invisible to the clients, which should only view the underlying modifications to the private heaps. This application required a generalization to indexed morphism families, and could also encode quiescence. In the Coq files, we have further verified a flat combiner and an in-place construction of a spanning tree of a graph.

Beyond the progress reported here, we expect that our notion of morphisms will have many other applications as well. In the immediate future, we plan to apply morphisms to procedures with linearization points whose placement in time can be determined only after the procedure's termination [8, 15]. Most related work deals with such programs by formalizing the dependence of the linearization points on the future events as a form of non-determinism, and the corresponding proofs employ features such as prophecy variables [1] (equivalently, speculations, backward simulations), which have not been reconciled with program lifting. It has recently been argued [4, 8] that future-dependence may not need non-determinism, as the placement of the linearization points can be deterministically resolved at the level of proofs. Thus, we expect that morphisms and FCSL will directly apply.

REFERENCES

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer Verlag, 2004.

[3] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.

[4] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving linearizability using forward simulations. In *CAV (to appear)*, 2017. Preliminary version available at http://arxiv.org/abs/1702.02705.

[5] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, 2007.

[6] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[7] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In *ECOOP*, 2014.

[8] Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Concurrent data structures linked in time. In *ECOOP*, 2017.

[9] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. Views: compositional reasoning for concurrent programs. In *POPL*, 2013.

[10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, 2010.

[11] Ilya Sergey et al. Programming and proving with distributed protocols. Personal communication.

[12] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.

[13] Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*, 2013.

[14] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, 2011.

[15] Prasad Jayanti. An optimal multi-writer snapshot algorithm. In *STOC*, 2005.

[16] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, 2015.

[17] Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, 2013.

[18] Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, 2013.

[19] Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, 2012.

[20] Peter Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report TR 25.085, IBM Laboratory Vienna, 1968.

[21] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, 2014.

[22] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Th. Comp. Sci.*, 375(1-3), 2007.

[23] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5), 1976.

[24] Gerhard Schellhorn, Heike Wehrheim, and John Derrick. How to prove algorithms linearisable. In *CAV*, 2012.

[25] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*, 2015.

[26] Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *OOPSLA*, 2016.
[27] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.
[28] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, 2013.
[29] The Coq Development Team. *The Coq Proof Assistant Reference Manual - Version V8.6*, 2016. http://coq.inria.fr/.
[30] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, 2013.

A GENERALIZED DEFINITIONS FOR INDEXED MORPHISM FAMILIES

In this appendix, we show how the definitions of morphism, f-stepping and f-stability, generalize to indexed families. When X is the unit type, we recover the morphism-related definitions from Section 3.

Definition A.1 (Indexed family of morphisms). An indexed family of morphisms $f : V \xrightarrow{X} W$ (or just family), consists of two components:

- A function from $x \in X$ to relation on the states of V and W, which we write as $(s_v, s_w) \in f\ x$, where s_v is a V-state, and s_w is a W-state.
- A function mapping $x \in X$ and an internal transition of V to internal transitions of W, which we write as $f\ x : V.\Delta_i \to W.\Delta_i$.

The components satisfy the following properties:

(1) *(W simulates V by internal steps)* if $t \in V.\Delta_i$ and $t\ s_v\ s'_v$ and $(s_v, s_w) \in f\ x$, then there exists x', s'_w such that $f\ x\ t\ s_w\ s'_w$ and $(s'_v, s'_w) \in f\ x'$.
(2) *(V simulates W by other steps)* if $s_w \xrightarrow[W]{}^* s'_w$ and $(s_v, s_w) \in f\ x$, then there exists s'_v such that $s_v \xrightarrow[V]{}^* s'_v$ and $(s'_v, s'_w) \in f\ x$.
(3) *(functionality)*: if $(s_{v1}, s_w) \in f\ x$ and $(s_{v2}, s_w) \in f\ x$, then $s_{v1} = s_{v2}$
(4) *(frame preservation)* there exists function $\phi : U_W \to U_v$ (notice the contravariance), such that: if $(s_v, s_w \rhd p) \in f\ x$, then $s_v = s'_v \rhd (\phi\ p)$ for some s', and $(s'_v \lhd \phi\ p, s_w \lhd p) \in f\ x$.
(5) *(other-fixity)* if $(s_v, s_w) \in f\ x$ and $(s'_v, s'_w) \in f\ x'$ and $a_o(s_w) = a_o(s'_w)$ then $a_o(s_v) = a_o(s'_v)$.
(6) *(index injectivity)* if $(s_v, s_{w1}) \in f\ x_1$ and $(s_v, s_{w2}) \in f\ x_2$ then $x_1 = x_2$

In most of the properties of Definition A.1, the index x is propagated unchanged. The only properties where x is significant are (1) and the new property (6). Compared to Definition 3.17, the property (1) allows that x changes into x' by a transition. In the Stack example in Section 5, if we lift e by using the index x = empty (i.e., write morph empty f e), then this index will evolve with e taking the transitions of Stack to track how e changes the self history by adding the entries for pushing 1 and 2. The property (6) requires that s_v uniquely determines the index x. In the Stack example, it is easy to see that the definition of f satisfies this property, because equal states have equal histories.

Definition A.2 (f-stepping). Let $f : V \xrightarrow{X} W$ be a family, and let s_w, s'_w be W-states. We say that x, s_w f-steps to x', s'_w, written $x, s \xrightarrow[f]{} x', s'$, if one of the following is true:

(1) exists $t \in V.\Delta_i$ and s_v, s'_v, such that $(s_v, s_w) \in f\ x$, $(s'_v, s'_w) \in f\ x'$, $t\ s_v\ s'_v$ and $f\ x\ t\ s_w\ s'_w$
(2) $s_w \xrightarrow[W]{} s'_w$

In other words, x, s_w steps by f into x', s'_w, either if it steps by ordinary interference on W, or the step is an f-image of a step by an internal transition in V. We write $\xrightarrow[f]{}^*$ for reflexive-transitive closure of $\xrightarrow[f]{}$.

Definition A.3 (f-stability). Let $f : V \xrightarrow{X} W$ be a family. A predicate P over X and W-states is *f-stable in state* x, s if whenever $x, s \underset{f}{\longrightarrow}^* x', s'$, then $P\ x'\ s'$. Predicate P is f-stable if it is f-stable in state x, s for every x, s for which $P\ x\ s$. Given a predicate P over X and W-states, we define its *f-stabilization* P^f as the following predicate:

$$P^f\ x\ s\ \widehat{=}\ \forall x'\ s'.\, x, s \underset{f}{\longrightarrow}^* x', s' \rightarrow P\ x'\ s'.$$

B DENOTATIONAL SEMANTICS

Our semantic model largely relies on the denotational semantic of *action trees* [17]. A tree implements a finite partial approximation of program behavior; thus a program of type ST $V\ A$ will be denoted by a set of such trees. The set may be infinite, as some behaviors may only be reached in the limit, after infinitely many finite approximations.

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

In this appendix, we first define the denotation of each of our commands as a set of trees. Then we define the semantic behavior for trees wrt. resource states, in a form of operational semantics for trees. Then we relate this low-level operational semantics of trees to high-level transitions of a resource by an always predicate (Section B) that ensures that a tree is resilient to any amount of interference, and that all the operational steps by a tree are *safe*. The always predicate will be instrumental in defining the vrf-predicate transformer from Section 3, and from there, in defining the type of Hoare triples $\{P\}\ A\ \{Q\}@V$. Both the ST $V\ A$ type and the Hoare triple type will be *complete lattices* of sets of trees, giving us a suitable setting for modeling recursion. The *soundness* of FCSL follows from showing that the lemmas about the vrf predicate transformer listed in Section 3, are satisfied by the denotations of the commands.

We choose the Calculus of Inductive Constructions (CiC) [2, 29] as our meta logic. This has several important benefits. First, we can define a *shallow embedding* of our system into CiC that allows us to program and prove directly *with the semantic objects*, thus immediately lifting to a full-blown programming language and verification system with higher-order functions, abstract types, abstract predicates, and a module system. We also gain a powerful dependently-typed λ-calculus, which we use to formalize all semantic definitions and meta theory, including the definition of action trees by *iterated inductive definitions* [29], specification-level functions, and programming-level higher-order procedures. Finally, we were able to mechanize the entire semantics and meta theory in the Coq proof assistant implementation of CiC.

Action trees and program denotations

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

$$
\begin{array}{rcl}
\mathsf{tree}\ V\ A & \widehat{=} & \mathsf{Unfinished} \\
& | & \mathsf{Ret}\ (v : A) \\
& | & \mathsf{Act}\ (a : \mathsf{action}\ V\ A) \\
& | & \mathsf{Seq}\ (T : \mathsf{tree}\ V\ B)\ (K : B \rightarrow \mathsf{tree}\ V\ A) \\
& | & \mathsf{Par}\ (T_1 : \mathsf{tree}\ V\ B_1)\ (T_2 : \mathsf{tree}\ V\ B_2)\ (K : B_1 \times B_2 \rightarrow \mathsf{tree}\ V\ A) \\
& | & \mathsf{Morph}\ (x : X)\ (f : W \xrightarrow{X} V)\ (T : \mathsf{tree}\ W\ A)
\end{array}
$$

Most of the constructors in Definition B.1 are self-explanatory. Since trees have finite depth, they can only approximate potentially infinite computations, thus the Unfinished tree indicates an incomplete approximation. Ret v is a terminal computation that returns value $v : A$. The constructor Act takes as a parameter an action a : action V A, as defined in Section 3. Seq T K sequentially composes a B-returning tree T with a continuation K that takes T's return value and generates the rest of the approximation. Par T_1 T_2 K is the parallel composition of trees T_1 and T_2, and a continuation K that takes the pair of their results when they join. CiC's iterated inductive definition permits the recursive occurrences of tree to be *nonuniform* (e.g., tree B_i in Par) and *nested* (e.g., the *positive* occurrence of tree A in the continuation). Since the CiC function space includes case-analysis, the continuation may branch upon the argument. The Morph constructor embeds an index $x : X$, morphism $f : W \xrightarrow{X} V$, and tree T : tree W A for the underlying computation. The constructor will denote T should be executed so that each of its actions is modified by f with an index x. We can now define the denotational model of our programs; that is the type ST V A of sets of trees, containing Unfinished.

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

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

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

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

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

By including Unfinished in all the denotations, we ensure that behaviors of the components are preserved in the composition. For example, the parallel composition {Unfinished} $\parallel$ e' is denoted by the set below which contains an image of each tree from e', thus capturing the behaviors of e'.

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

Operational semantics of action trees

The judgment for small-step operational semantics of action trees has the form $\Delta \vdash \bar{x}, s, T \xrightarrow{\pi} \bar{x}', s', T'$ (Figure 3). We explain the components of this judgment next.

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

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

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

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

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

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

Third, the components s and s' are W-states, and T, T' : tree V A, for some A. The meaning of the judgment is that a tree T, when executed in a state s, under the context of morphisms Δ produces a new state s' and residual tree T', encoding what is left to execute. The resource of the trees and the states disagree (the states use resource W, the trees use V), but the morphism context Δ relates them as follows. Whenever the head constructor of the tree is an action, the action will first be morphed by applying all the morphisms in Δ in order, to the transitions that constitute the head action, supplying along the way the projections out of x to the morphisms. This will produce a new index x' and an action on W-states, which can be applied to s to obtain s'.

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

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

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

In order to define the operational semantics on trees, we next require a few auxiliary notions. First, we need a function $\Delta(\bar{x})(t)$ that morphs an internal transition t of a resource V, into a transition of a resource W, by iterating the morphisms in the context Δ of resource type $V \longrightarrow W$, and passing along the elements out of the tuple $\bar{x}$ of type $(X_0, \cdots, X_n)$. The function is defined by induction on the structure of Δ, as follows.

$$\begin{array}{rcl}
(\cdot)\ ()\ (t) & \mathrel{\widehat{=}} & t \\
(f_0 : V_1 \xrightarrow{X_0} W, \Delta)\ (x_0, \bar{x})\ t & \mathrel{\widehat{=}} & f_0\ x_0\ (\Delta\ \bar{x}\ t)
\end{array}$$

That is, if Δ is the empty context, the index is empty tuple (). In that case, there is nothing to do, so we just return the transition t. Otherwise, we strip the first morphism f_0 from the context, and the first index component x_0, iterate the construction on the smaller context and index tuple, and apply f_0 x_0 to the result of the iterated construction.

Second, we need to have a similar iterative construction on states as well, which will transforms the states according to morphisms in Δ. We write unwind Δ t x s x' s' to denote that the the transition t of the resource V steps from the W-state s to W-state s' in the morphism context Δ.

$$\frac{\mathsf{unwind}\ \Delta\ (a\ v)\ \bar{x}\ s\ \bar{x}'\ s'}{\Delta \vdash \bar{x}, s, \mathsf{Act}\ a \stackrel{\mathsf{ChoiceAct}}{\longrightarrow} \bar{x}', s'; \mathsf{Ret}\ v} \qquad \frac{}{\Delta \vdash \bar{x}, s, \mathsf{Seq}\ (\mathsf{Ret}\ v)\ K \stackrel{\mathsf{SeqRet}}{\longrightarrow} \bar{x}, s, K\ v}$$

$$\frac{\Delta \vdash \bar{x}, s, T \stackrel{\pi}{\longrightarrow} \bar{x}, s', T'}{\Delta \vdash \bar{x}, s, \mathsf{Seq}\ T\ K \stackrel{\mathsf{SeqStep}\ \pi}{\longrightarrow} \bar{x}', s', \mathsf{Seq}\ T'\ K} \qquad \frac{}{\Delta \vdash \bar{x}, s, \mathsf{Par}\ (\mathsf{Ret}\ v_1)\ (\mathsf{Ret}\ v_2)\ K \stackrel{\mathsf{ParRet}}{\longrightarrow} \Delta \vdash \bar{x}, s, K\ (v_1, v_2)}$$

$$\frac{\Delta \vdash \bar{x}, s, T_1 \stackrel{\pi}{\longrightarrow} \bar{x}', s', T_1'}{\Delta \vdash \bar{x}, s, \mathsf{Par}\ T_1\ T_2\ K \stackrel{\mathsf{ParL}\ \pi}{\longrightarrow} \Delta \vdash \bar{x}', s', \mathsf{Par}\ T_1'\ T_2\ K} \qquad \frac{\Delta \vdash \bar{x}, s, T_2 \stackrel{\pi}{\longrightarrow} \bar{x}', s', T_2'}{\Delta \vdash \bar{x}, s, \mathsf{Par}\ T_1\ T_2\ K \stackrel{\mathsf{ParR}\ \pi}{\longrightarrow} \Delta \vdash \bar{x}', s', \mathsf{Par}\ T_1\ T_2'\ K}$$

$$\frac{}{\Delta \vdash \bar{x}, s, \mathsf{Morph}\ f\ y\ (\mathsf{Ret}\ v) \stackrel{\mathsf{MorphRet}}{\longrightarrow} \bar{x}, s, \mathsf{Ret}\ v} \qquad \frac{\Delta, f \vdash (\bar{x}, y), s, T \stackrel{\pi}{\longrightarrow} (\bar{x}', y'), s', T'}{\Delta \vdash \bar{x}, s, \mathsf{Morph}\ f\ y\ T \stackrel{\mathsf{MorphStep}\ \pi}{\longrightarrow} \bar{x}', s', \mathsf{Morph}\ f\ y'\ T'}$$

Fig. 3. Judgment $\Delta \vdash \bar{x}, s, T \stackrel{\pi}{\longrightarrow} \bar{x}', s', T'$, for operational semantics on trees, which reduces a tree with respect to the path π.

The notion is again defined by induction on the structure of Δ, as follows:

$$\begin{array}{rcl} \mathsf{unwind}\ \cdot\ t\ ()\ s\ ()\ s' & \widehat{=} & t\ s\ s' \\ \mathsf{unwind}\ (f_0 : V_1 \stackrel{X_0}{\longrightarrow} W, \Delta)\ t\ (x_0, \bar{x})\ s\ (x_0', \bar{x}')\ s' & \widehat{=} & \Delta\ (x_0, \bar{x})\ t\ s\ s' \wedge \exists s_1\ s_1'. (s_1, s) \in f_0\ x_0\ \wedge \\ & & \mathsf{unwind}\ \Delta\ \bar{x}\ s_1\ \bar{x}'\ s_1' \wedge (s_1', s') \in f_0\ x_0' \end{array}$$

If Δ is the empty context, there is nothing to do, and we just return $t\ s\ s'$. Otherwise, we require that s and s' are related by the image transition $\Delta\ (x_0, \bar{x})\ t$, but also that we can iteratively produce image states of s and s' under all the morphisms in the context.

We will frequently use the judgment in the case when Δ is the empty context, and correspondingly, $\bar{x}$ and $\bar{x}'$ are empty tuples (). In that case, we abbreviate, and write the judgment simply as

$$s, T \stackrel{\pi}{\longrightarrow} s', T'.$$

The operational semantics on trees in Figure 3 may not make a step on a tree for two different reasons. The first, benign, reason is that the the chosen path π does not actually determine an action or a redex in the tree T. For example, we may have T = Unfinished and π = ParR. But we can choose the right side of a parallel composition only in a tree whose head constructor is Par, which is not the case with Unfinished. We consider such paths that do not determine an action or a redex in a tree to be ill-formed. The second reason arises when π is actually well-formed. In that case, the constructors of the path uniquely determine a number of rules of the operational semantics that should be applied to step the tree. However, the premises of the rules may not be satisfies. For example, in the ChoiceAct rule, there may not exist a v such that unwind $\Delta\ (a\ v)\ \bar{x}\ s\ \bar{x}'\ s'$. To differentiate between these two different reasons, we first define the notion of well-formed, or *good* path, for a given tree.

Definition B.2 (Good paths and safety). Let T : tree V A and π be a path. Then the predicate good T π is defined as follows:

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

We now say that a state s is safe for the tree T and path π, written $s \in$ safe t π if:

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

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

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

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

Lemma B.3 (Coverage of stepping by transitions). *Let $\Delta : V \longrightarrow W$, and $\Delta \vdash \bar{x}, s, T \xrightarrow{\pi} \bar{x}', s', T'$. Then either the step corresponds to an idle transition (that is, $(\bar{x}, s) = (\bar{x}', s')$), or there exists a transition $a \in V.\Delta_i$, such that* unwind $\Delta\ a\ \bar{x}\ s\ \bar{x}'\ s'$.

Lemma B.4 (Other-fixity of stepping). *Let $\Delta : V \longrightarrow W$ and $\Delta \vdash \bar{x}, s, T \xrightarrow{\pi} \bar{x}', s', T'$. Then $a_o(s) = a_o(s')$.*

Lemma B.5 (S-preservation of stepping). *Let $\Delta : V \longrightarrow W$ and $\Delta \vdash \bar{x}, s, T \xrightarrow{\pi} \bar{x}', s', T'$. If $S.W(s)$ then $S.W(s')$.*

Lemma B.6 (Stability of stepping). *Let $\Delta : V \longrightarrow W$ and $\Delta \vdash \bar{x}, s, T \xrightarrow{\pi} \bar{x}', s', T'$. Then $s^\top \underset{W}{\longrightarrow^*} s'^\top$.*

Lemma B.7 (Determinism of stepping). *Let $\Delta : V \longrightarrow W$ and $\Delta \vdash \bar{x}, s, T \xrightarrow{\pi} \bar{x}', s', T'$, and $\Delta \vdash \bar{x}, s, T \xrightarrow{\pi} \bar{x}'', s'', T''$. Then $\bar{x}' = \bar{x}''$, $s' = s''$ and $T' = T''$.*

Lemma B.8 (Locality of stepping). *Let $\Delta : V \longrightarrow W$ and $\Delta \vdash \bar{x}, (s \rhd p), T \xrightarrow{\pi} \bar{x}', s', T'$. Then there exists s'' such that $s' = s'' \rhd p$, and $\Delta \vdash \bar{x}, (s \lhd p), T \xrightarrow{\pi} \bar{x}', (s'' \lhd p), T'$.*

Lemma B.9 (Safety monotonicity of stepping). *If $s \rhd p \in$ safe T π then $s \lhd p \in$ safe T π.*

Lemma B.10 (Framability of stepping). *Let $s \rhd p \in$ safe T π, and $s \lhd p, T \xrightarrow{\pi} s', T'$. Then there exists s'' such that $s' = s'' \lhd p$ and $s \rhd p, T \xrightarrow{\pi}, s'' \rhd p, T'$.*

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

Lemma B.11 (Stepping under morphism). *Let $f : V \xrightarrow{X} W$ and $(s_v, s_w) \in f\ x$. Then the following hold:*

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

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

Predicate transformers

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

Definition B.12. Let T : tree V A, and ζ be a sequence of paths. Also, let X be an assertion over V-states and V-trees, and Q be an assertion over A-values and V-states. We define the following predicate transformers:

$$\begin{array}{rcl}
\mathsf{always}^{\zeta}\ T\ X\ s & \widehat{=} & \text{if } \zeta = \pi :: \zeta' \text{ then } \forall s_2.\ s \xrightarrow[V]{}^{*} s_2 \rightarrow \\
& & \quad \mathsf{safe}\ T\ \pi\ s_2\ \wedge X\ s_2\ T \wedge \forall s_3\ T'.\ s_2, T \xrightarrow{\pi} s_3, T' \rightarrow \mathsf{always}^{\zeta'}\ T_2\ X\ s_3 \\
& & \text{else } \forall s_2.s \xrightarrow[V]{}^{*} s_2 \rightarrow X\ s_2\ T \\
\mathsf{always}\ T\ X\ s & \widehat{=} & \forall \zeta.\ \mathsf{always}^{\zeta}\ T\ X\ s \\
\mathsf{after}\ T\ Q & \widehat{=} & \mathsf{always}\ T\ (\lambda\, s'\ T'.\ \forall v.\ T' = \mathsf{Ret}\ v \implies Q\ v\ s')
\end{array}$$

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

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

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

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

$$\mathsf{vrf}\ e\ Q\ s\ \widehat{=}\ V.S\ s \wedge \forall T \in e.\ \mathsf{after}\ T\ Q\ s$$

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

$$V.S\ s \rightarrow P\ s \rightarrow \mathsf{vrf}\ e\ Q\ s$$

which is the formulation we used in Section 3 to define the Hoare triples.

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

Theorem B.13 (Soundness).

- *All the properties of* vrf *predicate transformer from Section 3 are valid.*
- *The sets* ST V A *and* $\{P\}$ A $\{Q\}$ *are complete lattices under subset ordering with the set* {Unfinished} *as the bottom. Thus one can compute the least fixed point of every monotone function by Knaster-Tarski theorem.*
- *All program constructors are monotone.*