Communicating State Transition Systems for Fine-Grained Concurrent Resources

Aleksandar Nanevski[1], Ruy Ley-Wild[2], Ilya Sergey[1], and Germán Andrés Delbianco[1]

[1] IMDEA Software Institute, Spain
{aleks.nanevski, ilya.sergey, german.delbianco}@imdea.org
[2] LogicBlox, USA
ruy.leywild@logicblox.com

Abstract. We present a novel model of concurrent computations with shared memory and provide a simple, yet powerful, logical framework for uniform Hoare-style reasoning about partial correctness of coarse- and fine-grained concurrent programs. The key idea is to specify arbitrary resource protocols as communicating *state transition systems* (STS) that describe valid states of a resource and the transitions the resource is allowed to make, including transfer of heap ownership. We demonstrate how reasoning in terms of communicating STS makes it easy to crystallize behavioral invariants of a resource. We also provide *entanglement* operators to build large systems from an arbitrary number of STS components, by interconnecting their lines of communication. Furthermore, we show how the classical rules from the Concurrent Separation Logic (CSL), such as scoped resource allocation, can be generalized to fine-grained resource management. This allows us to give specifications as powerful as Rely-Guarantee, in a concise, scoped way, and yet regain the compositionality of CSL-style resource management. We proved the soundness of our logic with respect to the denotational semantics of action trees (variation on Brookes' action traces). We formalized the logic as a shallow embedding in Coq and implemented a number of examples, including a construction of coarse-grained CSL resources as a modular composition of various logical and semantic components.

1 Introduction

There are two main styles of program logics for shared-memory concurrency, customarily divided according to the supported kind of granularity of program interference. Logics for coarse-grained concurrency such as Concurrent Separation Logic (CSL) [12, 14] restrict the interference to critical sections only, but generally lead to more modular specifications and simpler proofs of program correctness. Logics for fine-grained concurrency, such as Rely-Guarantee (RG) [8] admit arbitrary interference, but their specifications have traditionally been more monolithic, as we shall illustrate. In this paper, we identify the essential ingredients required for compositional specification of concurrent programs, and combine them in a novel way to reconcile the two approaches. We present a semantic model and a logic that enables specification and reasoning about fine-grained programs, but in the style of CSL. To describe our contribution more precisely, we first compare the relevant properties of CSL and RG.

CSL employs *shared resources* and associated *resource invariants* [13], to abstract the interference between threads. A resource r is a chunk of shared state, and a resource invariant I is a predicate over states, which holds of r whenever all threads are outside the critical section. By mutual exclusion, when a thread enters a critical section for r, it acquires ownership and hence exclusive access to r's state. The thread may mutate the shared state and violate the invariant I, but it must restore I before releasing r and leaving the critical section, as given by the following CSL rule [2].

$$\frac{\Gamma \vdash \{p * I\}\ c\ \{q * I\}}{\Gamma, r : I \vdash \{p\}\ \mathsf{with}\ r\ \mathsf{do}\ c\ \{q\}}\ \textsc{CritSecCSL}$$

Γ is a context of currently existing resources. The rule for parallel composition assumes that forked threads don't share any state beyond that of the resources in Γ, and may divide the private state of the parent thread disjointly among the children.

$$\frac{\Gamma \vdash \{p_1\}\ c_1\ \{q_1\} \qquad \Gamma \vdash \{p_2\}\ c_2\ \{q_2\}}{\Gamma \vdash \{p_1 * p_2\}\ c_1 \parallel c_2\ \{q_1 * q_2\}}\ \textsc{ParCSL}$$

A private heap of a thread may be promoted into a freshly allocated shared resource in a scoped manner by the following rule.

$$\frac{\Gamma, r : I \vdash \{p\}\ c\ \{q\}}{\Gamma \vdash \{p * I\}\ \mathsf{resource}\ r\ \mathsf{in}\ c\ \{q * I\}}\ \textsc{ResourceCSL}$$

One may see from these rules that resources are abstractions that promote modularity. In particular, one may verify a thread wrt. the smallest resource context required. By context weakening, the introduction of new resources will not invalidate the existing verification. Thread-local resources can be hidden from the environment by the ResourceCSL rule.

In RG, the interaction between threads is directly specified by the rule for parallel composition.[3]

$$\frac{R \vee G_2, G_1 \vdash \{p\}\ c_1\ \{q_1\} \qquad R \vee G_1, G_2 \vdash \{p\}\ c_2\ \{q_2\}}{R, G_1 \vee G_2 \vdash \{p\}\ c_1 \parallel c_2\ \{q_1 \wedge q_2\}}\ \textsc{ParRG}$$

The rely transition R and guarantee transitions G_1 and G_2 are relations on states. A rely specifies the thread's expectations of state transitions made by its environment. A guarantee specifies the state transitions made by the thread itself. The disjunctive combinations of R and G's in the rule captures the idea we call *forking shuffle*, whereby upon forking, the thread c_1 becomes part of the environment for c_2 and vice-versa.

RG is more expressive than CSL because transitions can encode arbitrary protocols on shared state, whereas CSL is specialized to a fixed mutual exclusion protocol on critical sections. But, CSL is more compositional in manipulating resources. Where a CSL resource invariant specifies the behavior of an individual chunk of shared state, the transitions in RG treat the whole state as monolithically shared. Feng's work on Local Rely Guarantee (LRG) [5] has made first steps in improving RG in this respect.

[3] In the presence of heaps, the rule is more complicated [6, 18], but we elide the issue here.

1.1 Contributions

We propose that a logic for fine-grained concurrency can be based on a notion of a *fine-grained resource*. Fine-grained resources serve as buildings blocks for program specification, and generalize CSL-style coarse-grained resource management. A fine-grained resource is specified by a resource invariant, as in CSL, but it also adds transitions in the form of relations between resource states. Thus, it is best viewed as a *state transition system* (STS), where the resource invariant specifies the state space. We identify a number of properties that an STS has to satisfy to specify a fine-grained resource, and refer to such STSs as *concurroids*. We refer to our generalization of CSL as Fine-grained CSL (FCSL).

There are two main ideas that we build on in FCSL, and which separate FCSL from LRG and other recent related work [4,15,17] (see Section 6 for details): (a) *subjectivity* and (b) *communication*. Subjectivity [10] means that each state of a concurroid STS describes not only the shared resource, but also *two* abstractions of it that represent the views of the state by the thread, and by its environment, respectively. Subjectivity will enable us to capture the idea of forking shuffle by a rule for parallel composition akin to PARCSL (but with a somewhat generalized notion of separating conjunction ($*$) [10]), rather than in the monolithic style of PARRG.

To compositionally build large systems out of a number of smaller ones, we make concurroids communicate. In addition to standard for STS *internal* transitions between states, concurroids contain *external* transitions. These may be thought of as "wires" whose one end is connected to a state in the STS, but whose other end is dangling, representing either an "input" into or an "output" out of the STS. Concurroids can be *entangled*, *i.e.*, composed by interconnecting their dangling wires of opposite polarity, where the interconnections serve to transfer heap ownership between concurroids. Communication and entanglement endow FCSL with the compositionality of CSL. For example, entanglement generalizes the notion of adding a resource to the context Γ in RESOURCECSL. We also rely on entanglement to formulate a rule generalizing the scoped resource allocation of RESOURCECSL. More precisely, our contributions are:

- We identify STSs with subjectively-shaped states (concurroids) and a number of algebraic properties, as a natural model for scalable concurrency verification. We show how communication enables composing larger STSs out of smaller ones.
- We present FCSL—a simple and expressive logic for fine-grained resources that combines expressivity of RG with the compositional resource management of CSL.
- We illustrate FCSL by showing how to implement a coarse-grained resource of CSL by a fine-grained resource of FCSL in which an explicit spin lock protects the resource's state. We also implemented examples such as ticketed locks, that go beyond coarse-grained CSL resources, and present them in the extended version [11].
- We implemented FCSL [11] as a shallow embedding within the type theory of the Calculus of Inductive Constructions (*i.e.*, Coq [1, 16]). Thus, FCSL naturally reconciles with features such as higher-order functions, abstract predicates, modules and functors. We formally instantiated the whole stack of abstractions: the semantic model is formalized in Coq, FCSL is built on top of the semantic model, CSL is built on top of FCSL, and then verified programs are built on top of CSL.

2 An Overview of Fine-Grained Resources

There are three different aspects along which fine-grained resources can be composed: space (*i.e.*, states), ownership, and time (*i.e.*, transitions). In this section, we describe how to represent these aspects in the assertion logic of FCSL.

Space The heap belonging to a fine-grained resource,[4] is explicitly identified by a *resource label*. We use assertions in the "points-to" style of separation logic, to name resources and identify their respective heaps. For example, the assertion

$$\ell_1 \overset{j}{\mapsto} h_1 * \ell_2 \overset{j}{\mapsto} h_2$$

describes a state in which the heaps h_1 and h_2 are associated with the resources labeled ℓ_1 and ℓ_2, respectively. The connective $*$ ensures that ℓ_1 and ℓ_2 are distinct labels, and that h_1 and h_2 are disjoint heaps. The superscript j indicates that the heaps are *joint* (shared), *i.e.*, can be accessed by any thread, even though they are owned by the resources ℓ_1 and ℓ_2, respectively.

The heaps h_1 and h_2 are not described by means of points-to assertions, but are built using operators for singleton heaps $x \to v$ and disjoint union $\cup$. For example, the heap of the resource lock, which explicitly encodes a coarse-grained resource with the resource invariant I [12] may be described by the assertion

$$\mathsf{lock} \overset{j}{\mapsto} ((lk \to b) \cup h) \wedge \mathsf{if}\ b\ \mathsf{then}\ h = \mathsf{empty}\ \mathsf{else}\ I\ h. \tag{1}$$

The assertion exposes the fact that the heap owned by lock contains a boolean pointer lk encoding a lock that protects the heap h. The conditional conjunct is a *pure* (*i.e.*, label-free) assertion, which describes an aspect of the ownership transfer protocol of CSL. When the lock is not taken (i.e. $b = \mathsf{false}$), the heap h satisfies the resource invariant. When the lock is taken, the heap is transfered to the private ownership of the locking thread, so h equals the empty heap, but lk remains in the ownership of lock.

Ownership Data in FCSL may be owned by a resource, as illustrated above, or by individual threads. The thread-owned data, however, is also associated with a resource, which it refines with *thread-relative* information. For example, the resource lock owns a pointer lk which operationally implements a lock. However, just knowing that the lock is taken or not is not enough for reasoning purposes; we need to know which thread has taken it, if any. Thus, we associate with each thread an extra bit of lock-related information, Own or ~~Own~~, which will identify the lock-owning thread as follows.

Following the idea of *subjectivity* [10], FCSL assertions are interpreted in a thread-relative way. We use *self* to name the interpreting thread, and *other* to name the combination of all other threads running concurrently with *self* (*i.e.*, the environment of *self*). We use two different assertions to describe thread-relative views: $\ell \overset{s}{\mapsto} v$ and $\ell \overset{o}{\mapsto} v$. The first is true in the *self* thread, if *self*'s view of the resource ℓ is v. The second is true in

[4] Or just resource for short. Later on, we explicitly identify CSL resources as *coarse-grained*.

the *self* thread, if *other*'s view of the resource ℓ is v. In this sense, the $\ell \overset{j}{\mapsto} v$ describes the resource's view of the data. In the case of lock, the thread that acquired the lock will validate the assertion:

$$\mathsf{lock} \overset{s}{\mapsto} \mathsf{Own} \wedge \mathsf{lock} \overset{j}{\mapsto} (lk \to \mathsf{true}),$$

while the symmetric assertion holds in all other threads at the same moment of time:

$$\mathsf{lock} \overset{j}{\mapsto} (lk \to \mathsf{true}) \wedge \mathsf{lock} \overset{o}{\mapsto} \mathsf{Own}.$$

In general, the values of the *self* and *other* views for *any resource* are elements of some resource-specific *partial commutative monoid* (PCM) [10]. A PCM is a set with a commutative and associative operation $\bullet$ with a unit element. $\bullet$ combines the *self* and *other* views into a view of the parallel composition of *self* and *other* threads. The $\bullet$ operation is commutative and associative because parallel composition of threads is commutative and associative, and the unit element models the view of the idle thread. Partiality models impossible thread combinations. For example, the elements of $O = \{\cancel{\mathsf{Own}}, \mathsf{Own}\}$ represent thread-relative views of the lock *lk*. O forms a PCM under the operation defined as $x \bullet \cancel{\mathsf{Own}} = \cancel{\mathsf{Own}} \bullet x = x$, with $\mathsf{Own} \bullet \mathsf{Own}$ undefined. The unit element is $\cancel{\mathsf{Own}}$, and the undefinedness of the last combination captures that two threads can't simultaneously own the lock. Notice that heaps form a PCM under disjoint union, with the empty heap as unit. Thus, they too obey the discipline required of the general *self* and *other* components.

Anticipating lock-related examples in Section 3, we combine thread-relative views of the lock with thread-relative views of the lock-protected heap *h*. We parametrize the resource lock by a PCM U, which the user may choose depending on the application. Then we use assertions over *pairs*, such as $\mathsf{lock} \overset{s}{\mapsto} (\mathsf{m_S}, \mathsf{a_S})$ and $\mathsf{lock} \overset{o}{\mapsto} (\mathsf{m_O}, \mathsf{a_O})$, to express that $\mathsf{m_S}, \mathsf{m_O} \in O$ are views of the lock *lk*, and $\mathsf{a_S}, \mathsf{a_O} \in U$ are views of the heap *h*. The following assertion illustrates how the different FCSL primitives combine. It generalizes (1) and defines the valid states of the resource lock.

$$\begin{array}{l} \mathsf{lock} \overset{s}{\mapsto} (\mathsf{m_S}, \mathsf{a_S}) \wedge \mathsf{lock} \overset{o}{\mapsto} (\mathsf{m_O}, \mathsf{a_O}) \wedge \mathsf{lock} \overset{j}{\mapsto} ((lk \to b) \uplus h) \wedge \\ \mathsf{if}\ b\ \mathsf{then}\ h = \mathsf{empty} \wedge \mathsf{m_S} \bullet \mathsf{m_O} = \mathsf{Own}\ \mathsf{else}\ I\ (\mathsf{a_S} \bullet \mathsf{a_O})\ h \wedge \mathsf{m_S} \bullet \mathsf{m_O} = \cancel{\mathsf{Own}} \end{array} \tag{2}$$

The assertion states that if the lock is taken ($b = \mathsf{true}$) then the heap *h* is given away, otherwise it satisfies the resource invariant I. In either case, the thread-relative views $\mathsf{m_S}$, $\mathsf{m_O}$, $\mathsf{a_S}$ and $\mathsf{a_O}$ are consistent with the resource's views of *lk* and *h*. Indeed, notice how $\mathsf{m_S}$, $\mathsf{m_O}$ and $\mathsf{a_S}, \mathsf{a_O}$ are first $\bullet$-joined (by the $\bullet$-operations of O and U, respectively) and then related to *b* and *h*; the former implicitly by the conditional, the latter explicitly, by the resource invariant I, which is now parametrized by $\mathsf{a_S} \bullet \mathsf{a_O}$.

Private heaps In addition to a private view of a resource, a thread may own a private heap as well. We describe such thread-private heaps by means of the same thread-relative assertions, but with a different resource label. We consider a dedicated resource for *private heaps*, with a dedicated label priv. Then we can write, say, $\mathsf{priv} \overset{s}{\mapsto} x \to 4$ to describe a heap consisting of a pointer x private to the *self* thread. By definition, $\mathsf{priv} \overset{j}{\mapsto} \mathsf{empty}$, *i.e.*, the *joint* heap of the priv resource is always empty.

Time Fine-grained reasoning requires characterization of the possible changes the threads can make to the state. We encode such a characterization as relations between states of possibly *multiple resources* (*i.e.*, using multiple labels). For example, coarse-grained resources require that upon successful acquisition, the resource's heap is transfered into the private ownership of the acquiring thread. In our fine-grained encoding, the transition can be represented as follows:

$$\begin{array}{l}\mathsf{priv} \overset{s}{\mapsto} h_S \quad * (\mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, a_S) \wedge \mathsf{lock} \overset{j}{\mapsto} ((lk \to \mathsf{false}) \cup h)) \rightsquigarrow \\ \mathsf{priv} \overset{s}{\mapsto} (h_S \cup h) * (\mathsf{lock} \overset{s}{\mapsto} (\mathsf{Own}, a_S) \wedge \mathsf{lock} \overset{j}{\mapsto} (lk \to \mathsf{true}))\end{array} \tag{3}$$

This transition preserves heap *footprints*, in the sense that the domain of the combined heaps in the source of the transition equals the domain in the target of the transition. We refer to such transitions as *internal*. Footprint preservation is an essential property, as it facilitates composing and framing transitions. In particular, adding additional labels and heaps with non-overlapping footprint to a source of an internal transition is guaranteed to produce non-overlapping footprints in the target of the transition as well.

We also consider *external* transitions that *can* acquire and release heaps. We use external transitions to build internal ones. For example, the above internal transition over priv and lock resources can be obtained as an interconnection (to be defined in Section 4) of two external transitions, each operating on an individual label.

$$\begin{array}{rcl}\mathsf{priv} \overset{s}{\mapsto} h_S & \overset{+h}{\rightsquigarrow} & \mathsf{priv} \overset{s}{\mapsto} (h_S \cup h) \\ \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, a_S) \wedge \mathsf{lock} \overset{j}{\mapsto} ((lk \to \mathsf{false}) \cup h) & \overset{-h}{\rightsquigarrow} & \mathsf{lock} \overset{s}{\mapsto} (\mathsf{Own}, a_S) \wedge \mathsf{lock} \overset{j}{\mapsto} lk \to \mathsf{true}\end{array} \tag{4}$$

The transition over priv takes a heap h as an input and attaches it to the *self* heap. The transition over lock gives the heap h as an output. When interconnected, the two transitions exchange the ownership of h between the lock and priv, producing (3).

A *concurroid* is an STS that formally represents a collection of resources. Each state of the STS contains a number of components, identified by the labels naming the individual resources. Each concurroid contains one internal transition, and an arbitrary number of external ones. The internal transition describes how threads specified by the concurroid may change their state in a single step. The external transitions are the "dangling wires", which provide means for composing different concurroids by *entangling* them, *i.e.*, interconnecting (some or all of) their dually polarized external transitions, to obtain a larger concurroid.

For example, if P is the concurroid for private heaps (containing a single label priv), and $L_{\{\mathsf{lock},lk,I\}}$ is the concurroid for a lock (with a single label lock, lock pointer lk and protected heap described by the coarse-grained resource invariant I), we could construct the entangled concurroid $CSL_{\{\mathsf{lock},lk,I\}} = P \bowtie L_{\{\mathsf{lock},lk,I\}}$ that captures the heap ownership-exchange protocol (3) of CSL for programs with *one coarse-grained resource*.[5] The entanglement can be iterated, to obtain an STS for *two coarse-grained resources* $CSL_{\{\mathsf{lock},lk,I\},\{\mathsf{lock}',lk',I'\}} = CSL_{\{\mathsf{lock},lk,I\}} \bowtie L_{\{\mathsf{lock}',lk',I'\}}$, and so on. In this way, concurroids generalize the notion of resource context from the RESOURCECSL rule, with entanglement modeling the addition of new resources to the context.

[5] The formal definition of the $\bowtie$ is postponed until Section 4.

Fig. 1 Semantics of selected FCSL assertions.

$$
\begin{array}{ll}
w \models \top & \text{iff always} \\
w \models \ell \overset{s}{\mapsto} v & \text{iff valid } w, \text{ and } w.s = \ell \to v \\
w \models \ell \overset{j}{\mapsto} h & \text{iff valid } w, \text{ and } w.j = \ell \to h \\
w \models \ell \overset{o}{\mapsto} v & \text{iff valid } w, \text{ and } w.o = \ell \to v \\
w \models p \wedge q & \text{iff } w \models p \text{ and } w \models q \\
w \models p * q & \text{iff valid } w, \text{ and } w = w_1 \cup w_2, \text{ and } w_1 \models p \text{ and } w_2 \models q \\
w \models p \mathrel{-\!\!*} q & \text{iff for every } w_1, \text{ valid } w \cup w_1 \text{ and } w_1 \models p \text{ implies } w \cup w_1 \models q \\
w \models p \circledast q & \text{iff valid } w, \text{ and } w.s = s_1 \cup s_2, \text{ and} \\
 & \quad [s_1 \mid w.j \mid s_2 \circ w.o] \models p \text{ and } [s_2 \mid w.j \mid s_1 \circ w.o] \models q \\
w \models \mathsf{this}\ w' & \text{if } w = w' \\
\models p \downarrow h & \text{iff for every valid } w,\ w \models p \text{ implies } \lfloor w \rfloor = h
\end{array}
$$

3 Reasoning with Concurroids

Auxiliary definitions A PCM-map is a finite map from labels (isomorphic to nat) to $\Sigma_{\mathbb{U}:\mathrm{pcm}}\mathbb{U}$. It associates each label with a pair of a PCM $\mathbb{U}$ and a value $v \in \mathbb{U}$. A heap-map is a finite map from labels to heaps. If m_1, m_2 are PCM-maps, then $m_1 \circ m_2$ is defined as $\mathsf{empty} \circ \mathsf{empty} = \mathsf{empty}$, and $((\ell \to_{\mathbb{U}} v_1) \cup m_1') \circ ((\ell \to_{\mathbb{U}} v_2) \cup m_2') = (\ell \to_{\mathbb{U}} v_1 \bullet v_2) \cup (m_1' \circ m_2')$, and undefined otherwise. By overloading the notation, we define state w as a triple $[s \mid j \mid o]$, where s, o are PCM-maps, and j is a heap-map. We abbreviate $[\ell \to v_s \mid \ell \to v_j \mid \ell \to v_o]$ with $\ell \to [v_s \mid v_j \mid v_o]$. w is valid if $w.s$, $w.j$, $w.o$ have the same domain as PCM-maps, $w.s \circ w.o$ is defined, and the heaps in $w.s$, $w.j$ and $w.o$ are disjoint (if $w.s$ and $w.o$ contain heaps in their codomain). State flattening $\lfloor w \rfloor$ is the disjoint union of all such heaps. $w_1 \cup w_2$ is the pairwise disjoint union of component maps of w_1 and w_2. The semantics of the main FCSL assertions is provided in Figure 1. The subjective assertions (*e.g.*, $w \models \ell \overset{s}{\mapsto} v$) constrain the value of one state component, assuming others to be existentially quantified over.

FCSL specifications take the form of Hoare 4-tuple $\{p\}\, c\, \{q\}@U$ expressing that the thread c has a precondition p, postcondition q, in a state space and under transitions defined by the concurroid U, which in FCSL takes the role of a resource context from CSL. We next present the characteristic inference rules of FCSL.

Parallel composition The rule for parallel composition in FCSL is similar to ParCSL, with Γ replaced by a concurroid U, which we will define formally in Section 4.

$$
\frac{\{p_1\}\, c_1\, \{q_1\}@U \qquad \{p_2\}\, c_2\, \{q_2\}@U}{\{p_1 \circledast p_2\}\, c_1 \parallel c_2\, \{q_1 \circledast q_2\}@U}\ \textsc{Par}
$$

The Par rule uses *subjective separating conjunction* $\circledast$ (see [10] and Figure 1) to split the state of $c_1 \parallel c_2$ into two. The split states contain the same labels, and equal *joint* portions, but the *self* and *other* portions are recombined to match the thread-relative views of c_1 and c_2. When the parent thread forks the children c_1 and c_2, the PCM values in the parent's *self* components are split between the children (similarly $*$ splits heaps in CSL), while the children's *other* component are implicitly induced to preserve overall

•-total (*i.e.*, c_1's *other* view includes c_2's *self* view, and vice versa). For example, in the case of one label ℓ, we have

$$\ell \overset{s}{\mapsto} a \bullet b \wedge \ell \overset{o}{\mapsto} c \implies (\ell \overset{s}{\mapsto} a \wedge \ell \overset{o}{\mapsto} c \bullet b) \circledast (\ell \overset{s}{\mapsto} b \wedge \ell \overset{o}{\mapsto} c \bullet a).$$

The implication encodes the idea of a forking shuffle from RG, but via states, rather than transitions as in RG. It allows us to use the *same* concurroid U to specify the transitions of both c_1 and c_2 in PAR, much like PARCSL uses the same context Γ. Essentially, we rely on the recombination of views to select the transitions of U available to each of c_1 and c_2, instead of providing distinct transitions for c_1 and c_2 as in PARRG.

We commonly encounter cases where the *other* views are existentially abstracted, hence the conjuncts $\ell \overset{o}{\mapsto} -$ are omitted. In those cases, we have the simplified bi-implication:

$$\ell \overset{s}{\mapsto} a \bullet b \iff \ell \overset{s}{\mapsto} a \circledast \ell \overset{s}{\mapsto} b \tag{5}$$

The implications generalize to ⊛-separated assertions with more than one distinct label.

We illustrate PAR and ⊛ with the example of concurrent incrementation [10,13] in a setting of a concurroid $CSL_{\mathsf{lock},lk,I}$ (*i.e.*, private state and one lock). The lock *lk* protects a shared integer pointer x, that is, the resource invariant is $I\ (a : nat)\ (h : heap) \mathrel{\widehat{=}} h = x \to a$. For the nat argument, we chose the PCM structure under addition; thus, an assertion $\mathsf{lock} \overset{s}{\mapsto} (-, \mathsf{a_S})$ expresses that the *self* thread has added $\mathsf{a_S}$ to x, and dually for $\mathsf{lock} \overset{o}{\mapsto} (-, \mathsf{a_O})$. Therefore, whenever the lock is not taken, x stores the sum $\mathsf{a_S} + \mathsf{a_O}$. This follows from interpreting • with + in the lock state invariant (2).

Procedure $\mathsf{incr}(n)$ acquires the lock to ensure exclusive access to x, increments x by n, and releases the lock. In FCSL, it has the following specification:

$$\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, 0)\right\}\ \mathsf{incr}(n)\ \left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, n)\right\} @ CSL_{\mathsf{lock},lk,I}$$

The specification states that incr runs in an empty private heap (hence by framing, in any larger heap), the lock is not owned by the calling thread initially, and will not be owned in the end. The addition of calling thread to x increases from 0 to n (hence by framing, from m to $m + n$). We now prove that $\mathsf{incr}(i) \parallel \mathsf{incr}(j)$ increments x by $i + j$.

$$\begin{array}{c}
\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, 0)\right\} \\
\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} \cup \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}} \bullet \cancel{\mathsf{Own}}, 0 + 0)\right\} \\
\left\{(\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, 0)) \circledast (\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, 0))\right\} \\
\begin{array}{cc}
\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, 0)\right\} & \left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, 0)\right\} \\
\mathsf{incr}(i) & \mathsf{incr}(j) \\
\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, i)\right\} & \left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, j)\right\}
\end{array} \\
\left\{(\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, i)) \circledast (\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, j))\right\} \\
\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, i + j)\right\}
\end{array}$$

The proof uses the bi-implication (5) to move between ⊛-separated assertions and •-joined views. The proof is compositional in the sense that the same verification of incr is used as a black box in both parallel threads, with the subproofs merely instantiating the parameter n with i and j respectively.

Injection The PAR rule requires c_1 and c_2 to share the same concurroid U, which describes the totality of their resources. If the threads use different concurroids, they first must be brought into a common entanglement, via the rule INJECT.

$$\frac{\{p\}\, c\, \{q\}@U \qquad r \text{ stable under } V}{\{p * r\}\ \mathsf{inject}\ c\ \{q * r\}@U \rightarrowtriangle V}\ \textsc{Inject}$$

If c is verified wrt. concurroid U, it can be *injected* (i.e. coerced) into a larger concurroid $U \rightarrowtriangle V$. In programs, we use the explicit coercion inject to describe the change of "type" from U to $U \rightarrowtriangle V$. Reading the rule bottom-up, it says we can ignore V, as V's transitions and c operate on disjointly-labeled state. V may change U's state by communication, but the change is bounded by U's external transitions. Thus, we are justified in verifying c against U alone. In this sense, INJECT may be seen as generalizing the rule for resource context weakening of CSL.

The connective $*$ splits the state according to labels of U and V; p and q describe the part labeled by U, and r describes the part labeled by V. Since r describes both the prestate and poststate, it has to be *stable* [11] under V; that is, determine a subset of V's states that remains fixed under transitions the *other* thread takes over the labels from V.

We illustrate INJECT and stability by verifying incr. To set the stage, we need atomic commands for reading from and writing to a pointer x. These have the following obvious specification relative to the concurroid P for private state:

$$\begin{array}{l}\left\{\mathsf{priv} \overset{s}{\mapsto} x \to v\right\}\ \ \mathsf{read}\ x\ \ \left\{\mathsf{priv} \overset{s}{\mapsto} x \to v \wedge \mathit{res} = v\right\}@P \\ \left\{\mathsf{priv} \overset{s}{\mapsto} x \to -\right\}\ \mathsf{write}\ x\ v\ \left\{\mathsf{priv} \overset{s}{\mapsto} x \to v\right\}@P\end{array}$$

The commands for acquiring and releasing lock exchange ownership of the protected pointer x. Thus, they have specifications relative to the concurroid $CSL_{\mathsf{lock},lk,I} = P \rightarrowtriangle L_{\mathsf{lock},lk,I}$, which we have already used before.

$$\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, 0)\right\}$$
$$\mathsf{acquire}$$
$$\left\{\exists \mathsf{a_O}.\mathsf{priv} \overset{s}{\mapsto} x \to \mathsf{a_O} * (\mathsf{lock} \overset{s}{\mapsto} (\mathsf{Own}, 0) \wedge \mathsf{lock} \overset{o}{\mapsto} (-, \mathsf{a_O}))\right\}@CSL_{\mathsf{lock},lk,I}$$

$$\left\{\mathsf{priv} \overset{s}{\mapsto} x \to \mathsf{a_S} + \mathsf{a_O} * (\mathsf{lock} \overset{s}{\mapsto} (\mathsf{Own}, 0) \wedge \mathsf{lock} \overset{o}{\mapsto} (-, \mathsf{a_O}))\right\}$$
$$\mathsf{release}$$
$$\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, \mathsf{a_S})\right\}@CSL_{\mathsf{lock},lk,I}$$

acquire assumes that lock is not taken, and that the *self* thread so far has added 0 to x. Thus, the overall contents of x is $0 + \mathsf{a_O} = \mathsf{a_O}$, where $\mathsf{a_O}$ is the addition of the *other* threads. Note that acquire does not have to be atomic:[6] as implemented, it just spins on lk, and after acquisition, x is transferred into the private heap of *self*. $\mathsf{a_O}$ must be existentially quantified, because *other*'s may add to x while acquire is spinning.

[6] The implementation of acquire and release relies on atomic actions (Section 5), specific for a particular concurroid, *e.g.* $CSL_{\mathsf{lock},lk,I}$.

release assumes that lock is taken by *self*, and that prior to taking lock, *self* and *other* have added 0 and $\mathsf{a_O}$ to x, respectively. After acquiring x, *self* has mutated it, so that its contents is $\mathsf{a_S} + \mathsf{a_O}$. After releasing, x is moved from the private heap to the *joint* portion of lock. The postcondition does not mention x, as once in *joint*, x's contents becomes unstable. Indeed, *other* may acquire the lock and change x after release terminates. However, *other* can't change the *self* view of x, which is now set to $\mathsf{a_S}$.

The following proof outline presents the implementation and verification of incr(n).

$$\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, 0)\right\}$$
acquire;
$$\left\{\exists \mathsf{a_O}.\mathsf{priv} \overset{s}{\mapsto} x \to \mathsf{a_O} * (\mathsf{lock} \overset{s}{\mapsto} (\mathsf{Own}, 0) \wedge \mathsf{lock} \overset{o}{\mapsto} (-, \mathsf{a_O}))\right\}$$
$res \leftarrow$ inject (read x);
$$\left\{\exists \mathsf{a_O}.\mathsf{priv} \overset{s}{\mapsto} x \to \mathsf{a_O} \wedge res = \mathsf{a_O} * (\mathsf{lock} \overset{s}{\mapsto} (\mathsf{Own}, 0) \wedge \mathsf{lock} \overset{o}{\mapsto} (-, \mathsf{a_O}))\right\}$$
inject (write x ($res + n$));
$$\left\{\exists \mathsf{a_O}.\mathsf{priv} \overset{s}{\mapsto} x \to n + \mathsf{a_O} * (\mathsf{lock} \overset{s}{\mapsto} (\mathsf{Own}, 0) \wedge \mathsf{lock} \overset{o}{\mapsto} (-, \mathsf{a_O}))\right\}$$
release
$$\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, n)\right\}$$

INJECT is used twice, to coerce read and write from the concurroid P to $CSL_{\mathsf{lock},lk,I}$. These commands manipulate the contents of priv, but retain the framing predicate $\mathsf{lock} \overset{s}{\mapsto} (\mathsf{Own}, 0) \wedge \mathsf{lock} \overset{o}{\mapsto} (-, \mathsf{a_O})$. This predicate is stable wrt. $L_{\mathsf{lock},lk,I}$. Intuitively, because *self* owns lock, *other* can't acquire x and add to it. Thus, no matter what *other* does, $\mathsf{a_O}$ and the framing predicate remain invariant.

To simplify the proof, we have not emphasized the invariance of $\mathsf{a_O}$ between calls to acquire and release, even though it is the case (we could do it using the rule EXIST from Figure 2). However, this invariance is what allowed us to calculate the contribution of *self* to x as n (*i.e.*, final contents of x minus $\mathsf{a_O}$). Without tracking $\mathsf{a_O}$, we would not know how much of the final contents of x is attributable to *self*, and how much to *other*.

Hiding refers to the ability to construct a concurroid V from the thread-private heap, in a scope of a thread c. The children forked by c can interfere on V's state, respecting V's transitions, but V is hidden from the environment of c. To the environment, V's state changes look like changes of the private heap of c. In this sense, hiding generalizes the RESOURCECSL rule to fine-grained resources.

$$\frac{\left\{\mathsf{priv} \overset{s}{\mapsto} h * p\right\} c \left\{\mathsf{priv} \overset{s}{\mapsto} h' * q\right\} @ (P \rtimes U) \rtimes V \qquad \text{(omitted side condition on } U \text{ and } V\text{)}}{\{\Psi\ g\ h * (\Phi(g) \mathrel{-\!\!*} p)\}\ \mathsf{hide}_{\Phi,g}\ c\ \{\exists g'.\Psi\ g'\ h' * (\Phi(g') \mathrel{-\!\!*} q)\} @ P \rtimes U}\ \text{HIDE}$$
$$\text{where } \Psi\ g\ h = \exists k{:}\mathsf{heap}.\, \mathsf{priv} \overset{s}{\mapsto} h \cup k \wedge \Phi(g) \downarrow k$$

Since installing V consumes a chunk of private heap, the rule requires the overall concurroid to support private heaps, *i.e.*, to be an entanglement $P \rtimes U$, where P is the concurroid for private heaps, and U is arbitrary (it is also possible to generalize the rule so as to be not tied to the specific concurroid P, see [11]). The omitted side condition on U and V is essential for the existence of entanglement and will be explained in Section 5. When U is of no interest, we set it to the empty concurroid E (Section 4), for which $P \rtimes E = P$.

In programs, we use the explicit coercion $\mathsf{hide}_{\Phi,g}$ to indicate the change of type from $(P \bowtie U) \bowtie V$ to $P \bowtie U$. The annotation $\Phi(g)$ corresponds to a set of *concrete states* of a concurroid V to be created. Its parameter g is a meaningful abstraction of such a set (*e.g.*, $(\mathsf{m_S}, \mathsf{a_S})$ for the $L_{\{\mathsf{lock},lk,I\}}$ concurroid) and can be thought of as an "abstract state". In the rule HIDE, g is the initial abstract state, *i.e.*, upon creation, the state of V satisfies $\Phi(g)$. In the premise of the HIDE rule, the predicates $\mathsf{priv} \overset{s}{\mapsto} -$ describe the behavior of c on the private heaps, while p and q describe the state of the labels belonging to U and V. In the conclusion, $\Psi\ g\ h$ and $\Psi\ g'\ h'$ map the abstract states g and g' into private heaps h and h'. This follows from the definition of Ψ, in which $\Phi(g) \downarrow k$ indicates that states satisfying $\Phi(g)$ *erase* to the private heap k (see Figure 1). Thus, changes that c imposes on abstract states, appear as changes to private heaps for $\mathsf{hide}_{\Phi,g}\ c$.

In the conclusion, the assertion $\Phi(g) \mathrel{-\!\!*} p$ states that attaching any state satisfying $\Phi(g)$ to the chunk of the initial state identified by the labels from U produces a state in which p holds, "compensating" for the component k in Ψ. That is, p corresponds to an abstract state g and c can be safely executed in such a state. The rule guarantees that if c terminates with a postcondition q, then q corresponds to some abstract state g'.

We illustrate the rule with a proof outline for program $\mathsf{hide}_{\Phi,g}\ (\mathsf{incr}(n))$. We show how to choose Φ and g so that the program implements the following functionality. It starts with only the concurroid P, and the private heap containing pointers lk and x. It locally installs $L_{\mathsf{lock},lk,I}$, which makes x a shared pointer, protected by the lock lk. It runs $\mathsf{incr}(n)$, after which the local concurroid is disposed, and lk and x return to the private heap. We prove that if initially $x \to 0$, then in the end $x \to n$. The abstract states are pairs $(\mathsf{m_S}, \mathsf{a_S})$, encodings of the *self* views of the concrete state of lock. Φ maps a *self* view into a predicate on the full state of lock, specifying *joint* and *other* views as well.

$$\begin{aligned}\Phi\ (\mathsf{m_S}, \mathsf{a_S}) = {} & \mathsf{lock} \overset{s}{\mapsto} (\mathsf{m_S}, \mathsf{a_S}) \wedge \mathsf{lock} \overset{o}{\mapsto} (\cancel{\mathsf{Own}}, 0) \wedge \\ & \mathsf{if}\ \mathsf{m_S} = \cancel{\mathsf{Own}}\ \mathsf{then}\ \mathsf{lock} \overset{j}{\mapsto} ((lk \to \mathsf{false}) \cup (x \to \mathsf{a_S}))\ \mathsf{else}\ \mathsf{lock} \overset{j}{\mapsto} (lk \to \mathsf{true})\end{aligned}$$

We choose the initial state $g = (\mathsf{m_S}, \mathsf{a_S}) = (\cancel{\mathsf{Own}}, 0)$: indicating that the lock is installed with lk unlocked, and x set to 0.

The proof outline uses the facts that $\Phi\ (\cancel{\mathsf{Own}}, \mathsf{a_S}) \downarrow lk \to \mathsf{false} \cup x \to \mathsf{a_S}$, and thus $\Psi\ (\cancel{\mathsf{Own}}, \mathsf{a_S})\ \mathsf{empty} = \mathsf{priv} \overset{s}{\mapsto} lk \to \mathsf{false} \cup x \to 0$. Also, $\Phi\ (\mathsf{m_S}, \mathsf{a_S}) \mathrel{-\!\!*} \mathsf{lock} \overset{s}{\mapsto} (\mathsf{m'_S}, \mathsf{a'_S})$ is equivalent to $(\mathsf{m_S}, \mathsf{a_S}) = (\mathsf{m'_S}, \mathsf{a'_S})$ in the label-free state.

$$\begin{array}{l}
\left\{\mathsf{priv} \overset{s}{\mapsto} lk \to \mathsf{false} \cup x \to 0\right\}@P \\
\{\Psi\ (\cancel{\mathsf{Own}}, 0)\ \mathsf{empty}\}@P \\
\left\{\Psi\ (\cancel{\mathsf{Own}}, 0)\ \mathsf{empty} * (\Phi\ (\cancel{\mathsf{Own}}, 0) \mathrel{-\!\!*} \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, 0))\right\}@P\ (= P \bowtie E) \\
\mathsf{hide}_{\Phi,(\cancel{\mathsf{Own}},0)}\ \left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, 0)\right\}@CSL_{\mathsf{lock},lk,I}\ (= P \bowtie E \bowtie L_{\mathsf{lock},lk,I}) \\
\qquad\qquad \mathsf{incr}(n) \\
\qquad\qquad \left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, n)\right\}@CSL_{\mathsf{lock},lk,I} \\
\left\{\exists g_2.\, \Psi\ g_2\ \mathsf{empty} * (\Phi\ g_2 \mathrel{-\!\!*} \mathsf{lock} \overset{s}{\mapsto} (\cancel{\mathsf{Own}}, n))\right\}@P \\
\{\Psi\ (\cancel{\mathsf{Own}}, n)\ \mathsf{empty}\}@P \\
\left\{\mathsf{priv} \overset{s}{\mapsto} lk \to \mathsf{false} \cup x \to n\right\}@P
\end{array}$$

The soundness of HIDE depends on a number of semantic properties of Φ [11]. The most important one is that states in the range of Φ have fixed *other* views for every label ℓ of V; equivalently, that environment threads for the program $\mathsf{hide}_{\Phi,g_1}\ c$ do not interfere with c on the states of V: all interference on V is *hidden* within the hide-section.

$$\text{if } w_1 \models \Phi\ g_1 \wedge (\ell \overset{o}{\mapsto} v_1 * \top) \text{ and } w_2 \models \Phi\ g_2 \wedge (\ell \overset{o}{\mapsto} v_2 * \top) \text{ then } v_1 = v_2$$

Concretely for our example, $\Phi\ g \wedge (\mathsf{lock} \overset{o}{\mapsto} v)$ implies $v = (\cancel{\mathsf{Own}}, 0)$, thus the above property clearly holds.

4 Concurroids Abstractly

A concurroid is a 4-tuple $V = (\mathcal{L}, \mathcal{W}, \tau, \mathcal{E})$ where: (1) $\mathcal{L}$ is a set of labels, where a label is a nat; (2) $\mathcal{W}$ is the *set of states*, each state $w \in \mathcal{W}$ having the structure described in Section 3; (3) τ is the *internal transition*, which is a relation on $\mathcal{W}$; (4) $\mathcal{E}$ is a set of pairs (α, ρ), where α and ρ are *external transitions* of V. An external transition is a function, mapping a heap h into a relation on $\mathcal{W}$. The components must satisfy a further set of requirements, discussed next.

State properties Every state $w \in \mathcal{W}$ is valid as defined in Figure 1, and its label footprint is $\mathcal{L}$, *i.e.* $\mathsf{dom}\ (w.s) = \mathsf{dom}\ (w.j) = \mathsf{dom}\ (w.o) = \mathcal{L}$. Additionally, $\mathcal{W}$ satisfies the property:

$$\begin{aligned}&\textit{Fork-join closure: } \forall t\text{:PCM-map}.\ w \triangleleft t \in \mathcal{W} \iff w \triangleright t \in \mathcal{W},\\ &\qquad\qquad \text{where } w \triangleleft t = [t \circ w.s \mid w.j \mid w.o], \text{and } w \triangleright t = [w.s \mid w.j \mid t \circ w.o]\end{aligned}$$

The property requires that $\mathcal{W}$ is closed under the realignment of *self* and *other* components, when they exchange a PCM-map t between them. Such realignment is part of the definition of $\circledast$, and thus appears in proofs whenever the rule PAR is used, *i.e.* whenever threads fork or join. Fork-join closure ensures that if a parent thread forks in a state from $\mathcal{W}$, then the child threads are supplied with states which also are in $\mathcal{W}$, and dually for joining.

Transition properties A concurroid transition γ is a relation on $\mathcal{W}$ satisfying:

$$\begin{aligned}&\textit{Guarantee: } (w, w') \in \gamma \implies w.o = w'.o\\ &\textit{Locality: } \quad \forall t\text{:PCM-map}.\ w.o = w'.o \implies (w \triangleright t, w' \triangleright t) \in \gamma \implies (w \triangleleft t, w' \triangleleft t) \in \gamma\end{aligned}$$

Guarantee restricts γ to only modify the *self* and *joint* components. Therefore, γ describes the behavior of a viewing thread in the subjective setting, but not of the thread's environment. In the terminology of Rely-Guarantee logics [5, 6, 8, 18], γ is a *guarantee* relation. To describe the behavior of the thread's environment, i.e. obtain a *rely* relation, we merely *transpose* the *self* and other components of γ:

$$\gamma^\top = \{(w_1^\top, w_2^\top) \mid (w_1, w_2) \in \gamma\},\ \text{where } w^\top = [w.o \mid w.j \mid w.s].$$

In this sense, FCSL transitions always encode *both* guarantee and rely relations.

Locality ensures that if γ relates states with a certain *self* components, then γ also relates states in which the *self* components have been simultaneously *framed* by a PCM-map t, *i.e.*, enlarged according to t. It thus generalizes the notion of locality from separation logic [14], with a notable difference. In separation logic, the frame t materializes out of nowhere, whereas in FCSL, t has to be appropriated from *other*; that is, taken out from the ownership of the environment.

An *internal* transition τ is a transition which is *reflexive* and preserves heap footprints. An *acquire* transition α, and a *release* transition ρ are functions mapping heaps to transitions which extend and reduce heap footprints, respectively, as formalized below. An external transition is either an acquire or a release transition. If $(\alpha, \rho) \in \mathcal{E}$, then α is an acquire transition, and ρ is a release transition.

$$\begin{array}{ll}
\textit{Footprint preservation:} & (w, w') \in \tau \implies \mathsf{dom}\ \lfloor w \rfloor = \mathsf{dom}\ \lfloor w' \rfloor \\
\textit{Footprint extension:} & \forall h{:}\mathsf{heap}.\ (w, w') \in (\alpha\ h) \implies \mathsf{dom}\ (\lfloor w \rfloor \cup h) = \mathsf{dom}\ \lfloor w' \rfloor \\
\textit{Footprint reduction:} & \forall h{:}\mathsf{heap}.\ (w, w') \in (\rho\ h) \implies \mathsf{dom}\ (\lfloor w' \rfloor \cup h) = \mathsf{dom}\ \lfloor w \rfloor
\end{array}$$

Internal transitions are reflexive so that programs specified by such transitions may be *idle* (*i.e.*, transition from a state to itself). Footprint preservation requires internal transitions to preserve the domains of heaps obtained by state flattening. Internal transitions may exchange the ownership of subheaps between the *self* and *joint* components, or change the contents of individual heap pointers, or change the values of non-heap (*i.e.*, auxiliary) state, which flattening erases. However, they cannot add new pointers to a state or remove old ones, which is the task of external transitions, as formalized by Footprint extension and reduction.

Example 1 (The concurroid for private state). $P = (\{\mathsf{priv}\}, \mathcal{W}_P, \tau_P, \{(\alpha_P, \rho_P)\})$, with

$$\begin{array}{lcl}
\mathcal{W}_P & = & \{\ \mathsf{priv} \to [h_\mathsf{S} \mid \mathsf{empty} \mid h_\mathsf{O}] \mid h_\mathsf{S} \text{ and } h_\mathsf{O} \text{ disjoint heaps}\ \}, \text{ and} \\
(w, w') \in \tau_P & \iff & w.s = \mathsf{priv} \to h_\mathsf{S}, w'.s = \mathsf{priv} \to h'_\mathsf{S}, \mathsf{dom}\ h_\mathsf{S} = \mathsf{dom}\ h'_\mathsf{S}, w.o = w'.o \\
(w, w') \in \alpha_P\ h & \iff & w.s = \mathsf{priv} \to h_\mathsf{S}, w'.s = \mathsf{priv} \to h_\mathsf{S} \cup h, w.o = w'.o \\
(w, w') \in \rho_P\ h & \iff & w.s = \mathsf{priv} \to h_\mathsf{S} \cup h, w'.s = \mathsf{priv} \to h_\mathsf{S}, w.o = w'.o
\end{array}$$

The internal transition admits arbitrary footprint-preserving change to the private heap h_S, while the acquire and release transitions simply add and remove the heap h from h_S.

Example 2 (The concurroid for a lock). $L_{\mathsf{lock},lk,I} = (\{\mathsf{lock}\}, \mathcal{W}_L, \tau_L, \{(\alpha_L, \rho_L)\})$, with $\mathcal{W}_L = \{\ w \mid w \models \text{ assertion (2)}\ \}$, and (assuming $w.o = w'.o$ everywhere):

$$\begin{array}{lcl}
(w, w') \in \tau_L & \iff & w = w' \\
(w, w') \in \alpha_L\ h & \iff & w.s = \mathsf{lock} \to (\mathsf{Own}, \mathsf{a}_\mathsf{S}),\ w.j = \mathsf{lock} \to (lk \to \mathsf{true}), \\
 & & w'.s = \mathsf{lock} \to (\cancel{\mathsf{Own}}, \mathsf{a}'_\mathsf{S}),\ w'.j = \mathsf{lock} \to ((lk \to \mathsf{false}) \cup h) \\
(w, w') \in \rho_L\ h & \iff & w.s = \mathsf{lock} \to (\cancel{\mathsf{Own}}, \mathsf{a}_\mathsf{S}),\ w.j = \mathsf{lock} \to ((lk \to \mathsf{false}) \cup h), \\
 & & w'.s = \mathsf{lock} \to (\mathsf{Own}, \mathsf{a}_\mathsf{S}),\ w'.j = \mathsf{lock} \to (lk \to \mathsf{true})
\end{array}$$

The internal transition admits no changes to the state w. The α_L transition corresponds to unlocking, and hence to the acquisition of the heap h. It flips the ownership bit from Own to ~~Own~~, the contents of the lk pointer from true to false, and adds the heap h to the resource state. The ρ_L transition corresponds to locking, and is dual to α_L. When locking, the ρ_L transition keeps the auxiliary view a_S unchanged. Thus, the resource

"remembers" the auxiliary view at the point of the last lock. Upon unlocking, the α_L transition changes this view into a'_S, where a'_S is some value that is coherent with the acquired heap h, *i.e.*, which makes the resource invariant $I\ (\mathsf{a}_S \bullet \mathsf{a}_O)\ h$ hold, and thus, the whole state belongs to $\mathcal{W}_L$.

Entanglement Let $U = (\mathcal{L}_U, \mathcal{W}_U, \tau_U, \mathcal{E}_U)$ and $V = (\mathcal{L}_V, \mathcal{W}_V, \tau_V, \mathcal{E}_V)$, be concurroids. The entanglement $U \rtimes V$ is a concurroid with the label component $\mathcal{L}_{U \rtimes V} = \mathcal{L}_U \cup \mathcal{L}_V$. The state set component combines the individual states of U and V by unioning their labels, while ensuring that the labels contain only non-overlapping heaps.

$$\mathcal{W}_{U \rtimes V} = \{w \cup w' \mid w \in \mathcal{W}_U, w' \in \mathcal{W}_V, \text{and } \lfloor w \rfloor \text{ disjoint from } \lfloor w' \rfloor\}$$

To define the transition components of $U \rtimes V$, we first need the auxiliary concept of transition interconnection. Given transitions γ_U and γ_V over $\mathcal{W}_U$ and $\mathcal{W}_V$, respectively, the interconnection $\gamma_1 \bowtie \gamma_2$ is a transition on $\mathcal{W}_{U \rtimes V}$ which behaves as γ_U (resp. γ_V) on the part of the states labeled by U (resp. V).

$$\gamma_1 \bowtie \gamma_2 = \{(w_1 \cup w_2, w'_1 \cup w'_2) \mid (w_i, w'_i) \in \gamma_i, w_1 \cup w_2, w'_1 \cup w'_2 \in \mathcal{W}_{U \rtimes V}\}.$$

The internal transition of $U \rtimes V$ is defined as follows, where id_U is the diagonal of $\mathcal{W}_U$.

$$\tau_{U \rtimes V} = (\tau_U \bowtie \mathsf{id}_V) \cup (\mathsf{id}_U \bowtie \tau_V) \cup \bigcup_{h, (\alpha_U, \rho_U) \in \mathcal{E}_U, (\alpha_V, \rho_V) \in \mathcal{E}_V} (\alpha_U\ h \bowtie \rho_V\ h) \cup (\alpha_V\ h \bowtie \rho_U\ h).$$

Thus, $U \rtimes V$ steps internally whenever U steps and V stays idle, or when V steps and U stays idle, or when there exists a heap h which U and V exchange ownership over by synchronizing their external transitions.

Example 3. The transitions α_p of P and ρ_L of $L_{\mathsf{lock},lk,I}$ have already been described in display (4) of Section 2, but using assertions, rather than semantically. The display (3) of Section 2 presents the interconnection $\alpha_P\ h \bowtie \rho_L\ h$, which moves h from $L_{\mathsf{lock},lk,I}$ to P, and is part of the definition of $\tau_{P \rtimes L_{\mathsf{lock},lk,I}}$. The latter further allows moving h in the opposite direction ($\alpha_L\ h \bowtie \rho_P\ h$), independent stepping of P ($\tau_P \bowtie \mathsf{id}_L$) and of $L_{\mathsf{lock},lk,I}$ ($\mathsf{id}_P \bowtie \tau_L$).

The external transitions of $U \rtimes V$ are those of U, framed wrt. the labels of V.

$$\mathcal{E}_{U \rtimes V} = \{(\lambda h.\, (\alpha_U\ h) \bowtie \mathsf{id}_V, \lambda h.\, (\rho_U\ h) \bowtie \mathsf{id}_V) \mid (\alpha_U, \rho_U) \in \mathcal{E}_U\}$$

We note that $\mathcal{E}_{U \rtimes V}$ somewhat arbitrarily chooses to frame on the transitions of U rather than those of V. In this sense, the definition interconnects the external transitions of U and V, but it keeps those of U "open" in the entanglement, while it "shuts down" those of V. The notation $U \rtimes V$ is meant to symbolize this asymmetry. The asymmetry is important for our example of encoding CSL resources, as it enables us to iterate the (non-associative) addition of new resources as $((P \rtimes L_{\mathsf{lock}_1,lk_1,I_1}) \rtimes L_{\mathsf{lock}_2,lk_2,I_2}) \rtimes \cdots$ while keeping the external transitions of P open to exchange heaps with new resources.

Clearly, many ways exist to interconnect transitions of two concurroids and select which transitions to keep open. In our implementation, we have identified several operators implementing common interconnection choices, and proved a number of equations

and properties about them (*e.g.*, all of them validate an instance of the INJECT rule). We also show a version of the INJECT rule with a different operator ($\ltimes$) [11]. However, as none of these operators is needed for the examples in this paper, we omit them.

Lemma 1. *$U \rtimes V$ is a concurroid.*

We can also reorder the iterated addition of lock concurroids.

Lemma 2 (Exchange law). $(U \rtimes V) \rtimes W = (U \rtimes W) \rtimes V$.

We close the section with the definition of the concurroid E which is the right unit of the entanglement operator $\rtimes$. E is defined as $E = (\emptyset, \mathcal{W}_E, id, \emptyset)$, where $\mathcal{W}_E$ contains only the empty state (i.e. the state with no labels).

5 Language and Logic

In the tradition of axiomatic program logics, the language of FCSL splits into purely functional expressions e (v when the expression is a value), and commands c with the effects of divergence, state and concurrency. We also include procedures F, for commands with arguments.

FCSL commands A command c satisfies the Hoare tuple $\{p\}\, c : A\, \{q\}@U$ if c's effect on states respects the internal transition of the concurroid U, c is *memory-safe* when executed from a state satisfying p, and concurrently with any environment that respects the transitions (internal and external) of U. Furthermore, if c terminates, it returns a value of type A in a state satisfying q. Formally, q may use a dedicated variable *res* of type A to name the return result.[7] FCSL uses a *procedure tuple*, $\forall x{:}B.\, \{p\}\, f\,(x) : A\, \{q\}@U$, to specify a potentially recursive higher-order procedure f taking an argument x of type B to a result of type A. The assertions p and q may depend on x. FCSL does not treat first-order looping commands, as these are special cases of recursive procedures. In the case of recursive procedures, p and q in the procedure tuple together correspond to a loop invariant, and typically are provided by the programmer.

The syntax of commands and procedures is as follows.

$$c ::= x \leftarrow c_1; c_2 \mid c_1 \parallel c_2 \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid F(e) \mid \mathsf{return}\ v \mid \mathsf{act}\ a \mid \mathsf{inject}\ c \mid \mathsf{hide}_{\Phi,g}\ c$$
$$F ::= f \mid \mathsf{fix}\ f.\, x.\, c$$

Commands and procedures include *atomic actions* act a, a monadic unit return v that returns v and terminates, a monadic bind (*i.e.* sequential composition) $x \leftarrow c_1; c_2$ that runs c_1 then substitutes its result v_1 for x to run c_2 (we write $c_1; c_2$ when $x \notin \mathsf{FV}(c_2)$), parallel composition $c_1 \parallel c_2$, a conditional, a procedure application $F(e)$, a procedure variable f, a fixed-point construct for recursion, and injection and hiding commands.

Judgments and inference rules The FCSL judgments are *hypothetical* under a context Γ that maps *program variables* x to their type and *procedure variables* f to their specification. We allow each specification to depend on the variables declared to the left.

$$\Gamma ::= \cdot \mid \Gamma, x{:}A \mid \Gamma, \forall x{:}B.\{p\} f(x) : A\, \{q\}@U$$

[7] When $A = \mathsf{unit}$, we suppress the type and the variable *res*, as we did in previous sections.

Fig. 2 FCSL inference rules.

$$\frac{\Gamma \vdash \{p\}\, c_1 : B\, \{q\}@U \qquad \Gamma, x : B \vdash \{[x/res]q\}\, c_2 : A\, \{r\}@U \qquad x \notin \mathsf{FV}(r)}{\Gamma \vdash \{p\}\, x \leftarrow c_1; c_2 : A\, \{r\}@U}\ \textsc{Seq}$$

$$\frac{\Gamma \vdash \{p_1\}\, c_1 : A_1\, \{q_1\}@U \qquad \Gamma \vdash \{p_2\}\, c_2 : A_2\, \{q_2\}@U}{\Gamma \vdash \{p_1 \circledast p_2\}\, c_1 \parallel c_2 : A_1 \times A_2\, \{[\pi_1\ res/res]q_1 \circledast [\pi_2\ res/res]q_2\}@U}\ \textsc{Par} \qquad \frac{\forall x{:}B.\, \{p\}\, f(x) : A\, \{q\}@U \in \Gamma}{\Gamma \vdash \forall x{:}B.\, \{p\}\, f(x) : A\, \{q\}@U}\ \textsc{Hyp}$$

$$\frac{\Gamma \vdash \{p_1\}\, c : A\, \{q_1\}@U \qquad \Gamma \vdash (p_1, q_1) \sqsubseteq (p_2, q_2)}{\Gamma \vdash \{p_2\}\, c : A\, \{q_2\}@U}\ \textsc{Conseq} \qquad \frac{\Gamma \vdash \{p\}\, c : A\, \{q\}@U \qquad r \text{ stable under } U}{\Gamma \vdash \{p \circledast r\}\, c : A\, \{q \circledast r\}@U}\ \textsc{Frame}$$

$$\frac{\Gamma \vdash \{e = \mathsf{true} \wedge p\}\, c_1 : A\, \{q\}@U \qquad \Gamma \vdash \{e = \mathsf{false} \wedge p\}\, c_2 : A\, \{q\}@U}{\Gamma \vdash \{p\}\, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : A\, \{q\}@U}\ \textsc{If}$$

$$\frac{\Gamma \vdash \{p_1\}\, c : A\, \{q_1\}@U \qquad \Gamma \vdash \{p_2\}\, c : A\, \{q_2\}@U}{\Gamma \vdash \{p_1 \wedge p_2\}\, c : A\, \{q_1 \wedge q_2\}@U}\ \textsc{Conj} \qquad \frac{\Gamma \vdash \{p\}\, c : A\, \{q\}@U \qquad \alpha \notin \mathsf{dom}\ \Gamma}{\Gamma \vdash \{\exists \alpha{:}B.\, p\}\, c : A\, \{\exists \alpha{:}B.\, q\}@U}\ \textsc{Exist}$$

$$\frac{\Gamma \vdash e : A \qquad p \text{ stable under } U}{\Gamma \vdash \{p\}\, \mathsf{return}\ e : A\, \{p \wedge res = e\}@U}\ \textsc{Ret} \qquad \frac{\Gamma, \forall x{:}B.\, \{p\}\, f(x) : A\, \{q\}@U, x{:}B \vdash \{p\}\, c : A\, \{q\}@U}{\Gamma \vdash \forall x{:}B.\, \{p\}\, (\mathsf{fix}\ f.\, x.\, c)(x) : A\, \{q\}@U}\ \textsc{Fix}$$

$$\frac{\Gamma \vdash \forall x{:}B.\, \{p\}\, F(x) : A\, \{q\}@U \qquad \Gamma \vdash e : B}{\Gamma \vdash \{[e/x]p\}\, F(e) : A\, \{[e/x]q\}@U}\ \textsc{App} \qquad \frac{\Gamma \vdash \{p\}\, c : A\, \{q\}@U \qquad r \text{ stable under } V}{\Gamma \vdash \{p * r\}\, \mathsf{inject}\ c : A\, \{q * r\}@U \rtimes V}\ \textsc{Inject}$$

$$\frac{\Gamma \vdash \left\{\mathsf{priv} \overset{s}{\mapsto} h * p\right\} c \left\{\mathsf{priv} \overset{s}{\mapsto} h' * q\right\}@(P \rtimes U) \rtimes V \qquad P, U \text{ and } V \text{ have disjoint sets of labels}}{\Gamma \vdash \{\Psi\ g\ h * (\Phi(g) \mathrel{-\!\!*} p)\}\, \mathsf{hide}_{\Phi,g}\ c\, \{\exists g'.\, \Psi\ g'\ h' * (\Phi(g') \mathrel{-\!\!*} q)\}@P \rtimes U}\ \textsc{Hide}$$

$$\text{where } \Psi\ g\ h = \exists k{:}\mathsf{heap}.\, \mathsf{priv} \overset{s}{\mapsto} h \cup k \wedge \Phi(g) \downarrow k$$

$$\frac{a = (U, A, \sigma, \mu) \text{ is an action} \qquad \Gamma \vdash (\sigma \wedge \mathsf{this}\ w, \lambda w'.\, (w, w', res) \in \mu) \sqsubseteq (p, q) \qquad p, q \text{ stable under } U}{\Gamma \vdash \{p\}\, \mathsf{act}\ a : A\, \{q\}@U}\ \textsc{Action}$$

Γ does not bind logical variables. In first-order Hoare logics, logical variables are implicitly universally quantified with global scope. In FCSL, we limit their scope to the Hoare tuples in which they appear. This is required for specifying recursive procedures, where a logical variable may be instantiated differently in each recursive call [9]. We also assume a formation requirement on Hoare tuples $\mathsf{FLV}(p) \supseteq \mathsf{FLV}(q)$, *i.e.*, that all free logical variables of the postcondition also appear in the precondition.

The inference rules of the Hoare tuple judgments for commands and procedures are presented in Figure 2. We note that the assertions and the annotations in the rules (*e.g.*, Φ in the HIDE rule) may freely use the variables in Γ. To reduce clutter, we silently assume the checks that all such specification level-entities are well-typed in their respective contexts Γ.

We have already discussed PAR, INJECT and HIDE rules in their versions where the return type $A = \mathsf{unit}$. The generalization to arbitrary A is straightforward. A side condition of HIDE ensures that the sets of labels of P, U and V don't clash, so the entanglement $(P \rtimes U) \rtimes V$ is defined. The rule FRAME is a special case of PAR when c_2 is taken to be the idle thread (*i.e.*, $c_2 = \mathsf{return}()$). Just like in the rule RET, we need to prove the framing assertion r stable, to account for the interference of the *other* threads. The rule FIX requires proving a Hoare tuple for the procedure body, under a

hypothesis that the recursive calls satisfy the same tuple. The procedure APPlication rule uses the typing judgment for expressions $\Gamma \vdash e : A$, which is the customary one from a typed λ-calculus, so we omit its rules; in our formalization in Coq, this judgment will correspond to the CiC's typing judgment. The CONSEQ rule uses the judgment $\Gamma \vdash (p_1, q_1) \sqsubseteq (p_2, q_2)$, which generalizes the customary side conditions $p_2 \implies p_1$ for strengthening the precondition and $q_1 \implies q_2$ for weakening the postcondition, to deal with the local scope of logical variables [11]. The other rules are standard from Hoare logic, except the ACTION rule for *atomic actions*. We devote the rest of the section to it.

Atomic actions Actions perform atomic steps from state to state, such as, *e.g.*, realigning the boundaries between, or changing the contents of *self*, *joint* and *other* state components. The actions thus serve to *synchronize* the changes to operational state (*i.e.*, heaps), with changes to the logical information required for verification (*i.e. auxiliary*, or *abstract*, parts of the state: $\mathsf{a_S}$, $\mathsf{a_O}$, *etc.*). If the logical information is erased, that is, if the states are flattened to heaps, then an action implements a single atomic memory operation such as looking up or mutating a heap pointer, CAS-ing over a heap pointer, or performing some other atomic *Read-Modify-Write* operation [7, § 5.6]. How an action manipulates the logical state is up to the user, depending on the application: we provide a formal definition of actions, and require that user's choices adhere to the definition.

An action is a 4-tuple $a = (U, A, \sigma, \mu)$ where: (1) the concurroid U whose internal transition a respects, (2) the type A of the action's return value, (3) the predicate σ on states describing the states in which the action could be executed, and (4) the relation μ relating the initial state, the ending state, and the ending result of the action. σ and μ are given in a large-footprint style, giving fully the heaps and the auxiliaries they accept.

For example, consider the action release used in Section 3 to release a lock and transfer the pointer x from a private heap of a thread to the ownership of the lock resource. This action is over the entangled concurroid $CSL_{\mathsf{lock},lk,I} = P \bowtie L_{\mathsf{lock},lk,I}$ as it transfers the ownership of $(x \to -)$. Its return value type is $A = \mathsf{unit}$. It can be executed in states in which the lock is taken by the *self* thread, and the pointer x is in the private heap. The contents of x is $\mathsf{a_S} + \mathsf{a'_S} + \mathsf{a_O}$, for some $\mathsf{a_S}$ and $\mathsf{a'_S}$, so that once x is transfered to the ownership of the lock resource, it satisfies the resource invariant. Thus:

$$
\begin{array}{lll}
w \in \sigma & \iff & w = \mathsf{priv} \to [x \to (\mathsf{a_S} + \mathsf{a'_S} + \mathsf{a_O}) \cup h_\mathsf{S} \mid \mathsf{empty} \mid h_\mathsf{O}] \cup \\
& & \quad \mathsf{lock} \to [(\mathsf{Own}, \mathsf{a'_S}) \mid lk \to \mathsf{true} \mid (\cancel{\mathsf{Own}}, \mathsf{a_O})] \\
(w, w', \mathit{res}) \in \mu & \iff & w = \mathsf{priv} \to [x \to (\mathsf{a_S} + \mathsf{a'_S} + \mathsf{a_O}) \cup h_\mathsf{S} \mid \mathsf{empty} \mid h_\mathsf{O}] \cup \\
& & \quad \mathsf{lock} \to [(\mathsf{Own}, \mathsf{a'_S}) \mid lk \to \mathsf{true} \mid (\cancel{\mathsf{Own}}, \mathsf{a_O})] \wedge \\
& & w' = \mathsf{priv} \to [h_\mathsf{S} \mid \mathsf{empty} \mid h_\mathsf{O}] \cup \\
& & \quad \mathsf{lock} \to [(\cancel{\mathsf{Own}}, \mathsf{a_S} + \mathsf{a'_S}) \mid lk \to \mathsf{false} \cup x \to (\mathsf{a_S} + \mathsf{a'_S} + \mathsf{a_O}) \mid (\cancel{\mathsf{Own}}, \mathsf{a_O})]
\end{array}
$$

Once the states are flattened into heaps, the σ and μ components of release reduce to describing the behavior of a memory mutation on the pointer lk. For example, the relation $\lfloor \mu \rfloor = \{(\lfloor w \rfloor, \lfloor w' \rfloor, r) \mid (w, w', r) \in \mu\}$ relates (h, h', r) iff

$$
\begin{array}{l}
h = (x \to (\mathsf{a_S} + \mathsf{a'_S} + \mathsf{a_O})) \cup h_\mathsf{S} \cup (lk \to \mathsf{true}) \cup h_\mathsf{O} \\
h' = (x \to (\mathsf{a_S} + \mathsf{a'_S} + \mathsf{a_O})) \cup h_\mathsf{S} \cup (lk \to \mathsf{false}) \cup h_\mathsf{O}
\end{array}
$$

Thus, operationally, release can be implemented as a single mutation to the lk pointer.

The inference rule ACTION takes an action $a = (U, A, \sigma, \mu)$ and checks that a satisfies that σ can be strengthened into p and μ can be weakened into q. As μ is not a

postcondition itself, but a relation taking input states, we first introduce a fresh logical variable w to name the input state using a predicate this. Then the predicate expressing post states for the action is computed out of μ and w, and it is this predicate that's weakened into q. p and q must be stable wrt. U, in order to account for the possibility that an interference of the environment appears just before, or just after, the action is executed.

Soundness and Implementation We have established the soundness of FCSL by exhibiting a denotational model based on *action trees* [10, 11], which are a variation on Brookes' action trace semantics, so we can formulate the following theorem.

Theorem 1. *FCSL is sound with respect to the denotational model of action trees.*

We developed the model in the logics of Calculus of Inductive Constructions, thus, the model is a shallow embedding in Coq, and its implementation is available on-line [11]. The implementation also defines denotational semantics for constructs and ascribes them types corresponding to rules in Figure 2. These type ascriptions require proofs, and together establish soundness of the logic, although rules/types in the implementation differ somewhat from those in Figure 2, facilitating encoding in Coq: (1) they use binary postconditions, (2) pre-/postconditions are in higher-order logic over heaps and PCMs, instead of notation from Figure 1, (3) they infer weakest-pre-/strongest-postconditions and (4) assertions are stabilized. The correspondence between the implementation and Figure 2 is straightforward, but established by hand.

6 Related Work

FCSL builds on the previous work on subjective auxiliary state and SCSL logic [10]. The SCSL logic contained the distinction between *self* and *other* views, which was essential for compositional implementation of *auxiliary state*. However, it contained exactly one coarse-grained resource, with no ability to create and dispose new resources. In contrast, FCSL can introduce any number of fine-grained resources in a scoped way.

The work on Concurrent Abstract Predicates (CAP) [4] introduces a notion of *shared region* that serves a similar purpose as concurroids, in that regions circumscribe a chunk of shared heap with a protocol governing its evolution. A *protocol* is defined by a set of atomic actions, which are RG-style transitions on private state and a region. In addition to heaps, regions may contain abstract capabilities that identify enabled actions. Thus there is a subtle mutual recursion in a protocol definition between an action and the capability to perform the action. A recurring pattern for this approach is quantification over *all* possible capabilities and placing them in a shared region, to be used up if needed in the execution of the protocol. The CAP framework could atomically change only one region; a restriction lifted in the recent work on Views [3] and HOCAP [15] that introduced *view shifts* to synchronize changes in several regions. Once allocated, CAP's regions have dynamically-scoped lifetime, and they can be disposed by a particular thread if it collects all corresponding region's capabilities. To the best of our knowledge, HOCAP does not allow the removal or scoped hiding of a shared region.

In contrast with CAP and their successors, FCSL does not require capabilities to perform actions, as these are naturally represented in the *self* and *other* views associated with a resource (and can also be seen as auxiliary state). Such auxiliary state is simpler

than capabilities; it is not subject to ownership transfer, and there is no need to quantify over all capabilities. In our experience, this simplicity extends to the specification of invariants and transitions, and to the proofs of stability. In FCSL, synchronizing changes over a number of concurroids is achieved directly at the level of transitions by means of entanglement, and at the level of programs by allowing actions to be defined over any concurroid, including entangled ones. Thus, no view shifts are required. The burden of stability proofs is further reduced in FCSL by formulating private heaps as a separate concurroid that one may, but need not, entangle with. Thus, when an action manipulates only the internal state of a resource, the attendant stability proofs can ignore private heaps, *e.g.*, the take action of a ticketed lock [4, 11]. Moreover, the communication in FCSL makes it possible for concurroids to pass heaps between each other directly, rather than going through private state. While the current paper does not present examples that exploit this ability, we have found it useful when verifying in FCSL a more advanced example of readers-writers, which we will present in future work.

CaReSL [17] uses the same notion of shared region as CAP, though it specifies the transitions in a manner closer to FCSL, namely by means of STS's. CaReSL does not directly provide subjective *self* and *other* views of a resource, but it provides a notion of *tokens*, whose ownership is exchanged between a thread and its environment. CaReSL assertions explicitly allow statements only about self-owned tokens, not *other*-owned ones. Thus, reasoning about the lack of logical changes to environment-owned data has to be encoded with a level of indirection, potentially quantifying over all tokens, similar to CAP's quantification over capabilities. A frequent side condition in CaReSL rules is that various assertions are *token-pure*, which does not have a direct correspondent in FCSL. Similar to CAP, CaReSL currently allows actions that work over only a single region, and will require an extension akin to view shifts to enable synchronized updates. CaReSL does not consider removal or scoped hiding of shared regions, although it can be emulated by introducing an empty "final" protocol state. Instead of stability checks in FCSL, in CaReSL one may stabilize assertions by composing them with environment stepping. In our experience, this does not change the proofs: the same obligations reappear in proofs out of stabilized hypotheses. On the other hand, CaReSL can reason about fine-grained data structure by means of refinement (a generalization of linearizability). FCSL supports higher-order functions by means of shallow embedding into CiC [1, 16], but we have not considered linearizability so far, which is future work.

Feng's Local Rely-Guarantee (LRG) [5] is, to the best of our knowledge, the first work that reconciled fine-grained reasoning in the style of RG with framing and hiding at the level of transitions (similar to our INJECT and HIDE). We differ from LRG in that we introduce communication and subjectivity into the mix; thus our injection and hiding rules take *self* and *other* views into account. The latter are a compositional form of auxiliary state, whereas LRG in practice has to use the classical, non-compositional form of auxiliary state [10, 13].

7 Conclusion and Future Work

We presented *concurroids*—a novel model for scalable shared-memory concurrency verification, based on communicating STS, and FCSL—a logic for concurroids.

In the future work, we are going to build a number of concurroids to encode common programming patterns. For example, dynamic allocation and deallocation of memory can be encoded via an allocator concurroid (without extensions of FCSL), and similarly for dynamic allocation and deallocation of locks. We hope to investigate if concurroids can be endowed with analogues of channel relabeling and restriction operators from process algebras, to provide finer control over interconnection and closure of external transitions. Finally, we plan to consider refinement which allows weakening the ascribed concurroid U of a program, to a coarser-grained concurroid V, if U can be shown to simulate V. One could then verify fine-grained concurrent ADTs against V, and afterwards hide the granularity by switching to U.

Acknowledgments We thank Anindya Banerjee, Thomas Dinsdale-Young and the ESOP 2014 anonymous reviewers for their comments. This research was partially supported by Spanish MINECO projects TIN2012-39391-C04-01 Strongsoft, TIN2010-20639 Paran10, AMAROUT grant PCOFUND-GA-2008-229599, and Ramon y Cajal grant RYC-2010-0743.

References

1. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.
2. S. Brookes. A semantics for concurrent separation logic. *Th. Comp. Sci.*, 375(1-3), 2007.
3. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL'13*.
4. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP 2010*.
5. X. Feng. Local rely-guarantee reasoning. In *POPL'09*.
6. X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP 2007*.
7. M. Herlihy and N. Shavit. *The art of multiprocessor programming*. M. Kaufmann, 2008.
8. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Prog. Lang. Syst.*, 5(4), 1983.
9. T. Kleymann. Hoare logic and auxiliary variables. *Formal Asp. Comput.*, 11(5), 1999.
10. R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL'13*.
11. A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Supporting Material. `http://software.imdea.org/~aleks/fcsl/`.
12. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Th. Comp. Sci.*, 375(1-3), 2007.
13. S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5), 1976.
14. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
15. K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP 2013*.
16. The Coq Development Team. *The Coq Proof Assistant Reference Manual - Version V8.4*, 2012. `http://coq.inria.fr/`.
17. A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP'13*.
18. V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007*.