Communicating State Transition Systems for Fine-Grained Concurrent Resources

Extended Version

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 appendix.
- 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* (see Appendix B) 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:

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

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

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

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

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.* $\mathit{CSL}_{\mathsf{lock},lk,I}$.

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 (see Appendix C). 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 ($\rtimes$) (see Appendix E). 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* $\mathsf{act}\ a$, a monadic unit $\mathsf{return}\ v$ 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. 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*.

Optional Appendices

In the optional appendices we describe in detail some technical insights behind FCSL and the model of concurroids. We also provide an extended overview of another case study, which was omitted in the paper body due to space constraints. The remainder of the paper is structured as follows. **Appendix A** describes a number of properties and requirements of the actions, presented in Section 5 of the main paper body. **Appendix B** formalizes the familiar notions of *Rely* and *Guarantee* relations, as well as a notion of *stability*, in terms of concurroids and their transitions. **Appendix C** lists the necessary properties of the abstraction functions Φ, used in the HIDE rule of FCSL. **Appendix D** defines Hoare ordering $(p_1, q_1) \sqsubseteq (p_2, q_2)$, used in the rule CONSEQ of the logic. **Appendix E** presents development of a large example: a concurroid for the ticketed lock algorithm. Finally, in **Appendix F** we present the denotational model of FCSL, based on *action trees* and *schedules*, and prove the main technical result: soundness of FCSL as a shallow embedding into the Calculus of Inductive Constructions.

A Abstract Properties of Actions

An action is a 4-tuple $a = (U, A, \sigma, \mu)$. U is a concurroid whose internal transition a is supposed to respect. A is the type of the return value of a. σ is the *safety predicate* (over states of U) describing under which conditions a may be executed. μ is the *stepping relation*; it relates the input state, output state, and the result of a.

Definition 1 (Action erasures). *Given an action a, the erasures $\lfloor\sigma\rfloor$ and $\lfloor\mu\rfloor$ of a's safety predicate and stepping relation are relations on heaps defined as follows.*

$$\begin{array}{ll} \lfloor w \rfloor \in \lfloor \sigma \rfloor & \iff w \in \sigma \\ (\lfloor w \rfloor, \lfloor w' \rfloor, r) \in \lfloor \mu \rfloor & \iff (w, w', r) \in \mu \end{array}$$

An *atomic* is a triple $\alpha = (A, \sigma, \mu)$. It's a special kind of actions, but over concrete heaps, rather than over states. States differ from heaps in that they are decorated with additional information such as auxiliary state and partitioning between *self*, *joint* and *other*. As with actions, A is the return type, σ is the safety predicate and μ is the stepping relation, but they all range over heaps.

We consider four different (parametrized classes of) atomics, corresponding to the four (parametrized) primitive memory operations that we consider.

Definition 2 (Primitive atomic actions).

$$\begin{array}{ll} \mathsf{Read}^{A}_{x} & = (A, (x \to_A -) \cup h, (x \to v) \cup h \rightsquigarrow (x \to v) \cup h \wedge \mathit{res} = v) \\ \mathsf{Write}\ x\ v & = (\mathsf{unit}, (x \to -) \cup h, (x \to -) \cup h \rightsquigarrow (x \to v) \cup h) \\ \mathsf{Skip} & = (\mathsf{unit}, h, h \rightsquigarrow h) \\ \mathsf{RMW}^{A\ B}_{x\ f\ g} & = (B, (x \to_A -) \cup h, (x \to v) \cup h \rightsquigarrow (x \to f(v)) \cup h \wedge \mathit{res} = g(v)) \end{array}$$

The last class $\mathsf{RMW}^{A\ B}_{x\ f\ g}$ corresponds to the family of *Read-Modify-Write* operations: they all atomically replace the current register value v with $f(v)$ for some pure function f, and return the result according to the function g [7, §5.6]. One particular

representative of this family is the CAS operation, which instantiates the parameters of RMW as follows:

$$\mathrm{CAS}_{A\ x\ v_1\ v_2} \ \widehat{=}\ \mathsf{RMW}^{A\ \mathsf{bool}}_{x\ f(v_1,v_2)\ g(v_1,v_2)}, \text{where}$$

$$\begin{aligned} f(v_1, v_2)(v) &= \mathsf{if}\ (v = v_1)\ \mathsf{then}\ v_2\ \mathsf{else}\ v_1 \\ g(v_1, v_2)(v) &= (v = v_1) \end{aligned}$$

Another representative of the RMW class is the atomic *fetch-and-increment* functions, which we will employ in Appendix E, defined as follows:

$$\mathrm{FAI}_x \ \widehat{=}\ \mathsf{RMW}^{\mathsf{nat}\ \mathsf{nat}}_{x\ \mathsf{inc}\ \mathsf{id}}, \text{where}$$

$$\mathsf{inc}(v) = v + 1 \quad \mathsf{id}(v) = v$$

Definition 3 (Operational action). *An action a is* operational *if its erasure corresponds to one of the atomics,* i.e.*, if there exists $b \in \left\{\mathsf{Read}^A_x, \mathsf{Write}\ x\ v, \mathsf{Skip}, \mathsf{RMW}^{A\ B}_{x\ f\ g}\right\}$ such that*

$$\lfloor \sigma_a \rfloor \subseteq \sigma_b \wedge \forall h \in \lfloor \sigma_a \rfloor\ h'\ r.\ (h, h', r) \in \lfloor \mu_a \rfloor \implies (h, h', r) \in \mu_b$$

In our examples we only consider operational actions, though the inference rules and the implementation in Coq don't currently enforce this requirement (the operationality of actions in the examples has been proved by hand).

Let $U = (\mathcal{L}, \mathcal{W}, \tau, \mathcal{E})$. The action $a = (U, A, \sigma, \mu)$ is required to satisfy the following properties.

$$\begin{aligned} \textit{Coherence} &: w \in \sigma \implies w \in \mathcal{W} \\ \textit{Safety monotonicity} &: w \triangleright t \in \sigma \implies w \triangleleft t \in \sigma \\ \textit{Step safety} &: (w, w', r) \in \mu \implies w \in \sigma \\ \textit{Internal stepping} &: (w, w', r) \in \mu \implies (w, w') \in \tau \\ \textit{Framing} &: w \triangleright t \in \sigma \implies (w \triangleleft t, w', r) \in \mu \implies \\ &\quad \exists w''.\ w' = w'' \triangleleft t \wedge (w \triangleright t, w'' \triangleright t, v) \in \mu \\ \textit{Erasure} &: \mathsf{defined}(\lfloor w \rfloor \cup h) \implies \lfloor w \rfloor \cup h = \lfloor w' \rfloor \cup h' \implies \\ &\quad (w, w_1, r) \in \mu \implies (w', w'_1, r') \in \mu \implies \\ &\quad r = r' \wedge \lfloor w \rfloor_1 \cup h = \lfloor w'_1 \rfloor \cup h' \\ \textit{Totality} &: \forall w.\ w \in \sigma \implies \exists w'\ v.\ (w, w', v) \in \mu \end{aligned}$$

The properties of Coherence, Step safety and Internal stepping are straightforward. Safety monotonicity states that if the action is safe in a state with a smaller *self* component (because the other component is enlarged by t), the action is also safe if we increase the *self* component by t.

Framing property says that if a steps in a state with a large *self* component $w \triangleleft t$, but is already safe to step in a state with a smaller *self* component $w \triangleright t$, then the result state and value obtained by stepping in $w \triangleleft t$ can be obtained by stepping in $w \triangleright t$, and moving t afterwards.

The Erasure property shows that the behavior of the action on the concrete input state obtained after erasing the auxiliary fields and the logical partition, doesn't depend on the erased auxiliary fields and the logical partition. In other words, if the input state have *compatible* erasures (that is, erasures which are subheaps of a common heap), then executing the action in the two states results in equal values, and final states that also

have compatible erasures. This is a standard property proved in concurrency logics that deal with auxiliary state and code [2, 13].

The Totality property shows that an action whose safety predicate is satisfied always produces a result state and value. It doesn't loop forever, and more importantly, it doesn't crash. We will use this property of actions in the semantics of programs to establish that if the program's precondition is satisfied, then all of the approximations in the program's denotation are either done stepping, or can actually make a step (i.e., they make progress).

So far, the actions are defined in a so-called *large footprint* style, *e.g.*, the definition of the action release in Section 5 mentions existentially-quantified heap h_S explicitly, To enable writing various actions in a *small footprint* style, we also enforce the property

$$\mathit{Locality} : w.o = w'.o \implies (w \triangleright t, w' \triangleright t, v) \in \mu \implies (w \triangleleft t, w' \triangleleft t, v) \in \mu$$

Curiously, if the default use of the logic is in a large footprint notation, then this property is not necessary as it is not used in any proofs.

B Self/Environment Stepping and Stability

Definition 4 (Self-stepping (Guarantee)).
Given a concurroid $U = (\mathcal{L}, \mathcal{W}, \tau, \mathcal{E})$, the environment stepping *relation $\mathcal{G}_U$ of U unions its internal and all external transitions of U, and then takes a transitive closure.*

$$\mathcal{G}_U = (\tau_U \cup \bigcup_{h,(\alpha,\rho)\in\mathcal{E}_X} (\alpha\ h) \cup (\rho\ h))^*.$$

Definition 5 (Environment stepping (Rely)).
Given a concurroid $U = (\mathcal{L}, \mathcal{W}, \tau, \mathcal{E})$, the environment stepping *relation $\mathcal{R}_U$ of U unions the transposed versions of the internal and all external transitions of U, and then takes a transitive closure.*

$$\mathcal{R}_U = (\tau^\top \cup \bigcup_{h,(\alpha,\rho)\in\mathcal{E}} (\alpha\ h)^\top \cup (\rho\ h)^\top)^*$$

Definition 6 (Stability). *An assertion (*i.e.*, a predicate over states) p is* stable *under concurroid U iff for every w, w' such that $(w, w') \in \mathcal{R}_U$, $w \models p \implies w' \models p$.*

C Properties of Φ Functions from the Hiding Rule

The abstraction function Φ is a user-specified annotation on the hide command. It maps values $g : G$ (where G is also user specified) to assertions, that is, predicates over states (equivalently, sets of states) of a concurroid V. For the soundness of the hiding rule, Φ is required to satisfy the following properties.

$$\begin{aligned}
&\mathit{Coherence} &&: w \in \Phi(g) \implies w \in \mathcal{W}_V\\
&\mathit{Injectivity} &&: w \in \Phi(g_1) \implies w \in \Phi(g_2) \implies g_1 = g_2\\
&\mathit{Surjectivity} &&: w_1 \in \Phi(g_1) \implies w_2 \in \mathcal{W}_W \implies w_1.o = w_2.o \implies \exists g_2.\, w_2 \in \Phi(g_2)\\
&\mathit{Guarantee} &&: w_1 \in \Phi(g_1) \implies w_2 \in \Phi(g_2) \implies w_1.o = w_2.o\\
&\mathit{Precision} &&: w_1 \in \Phi(g) \implies w_2 \in \Phi(g) \implies \lfloor w_1 \rfloor \cup h_1 = \lfloor w_2 \rfloor \cup h_2 \implies w_1 = w_2
\end{aligned}$$

Coherence and Injectivity are obvious. Surjectivity states that for every state w_2 of the concurroid W one can find an image g, under the condition that the *other* component of w_2 is well-formed according to Φ (typically, that the *other* component is equal to the unit of the PCM-map monoid for W). Guarantee is the property stated in the paper at the end of Section 3, but through assertions, rather than semantically. This property formalizes that environment of hide can't interference on V, as V is installed locally. Thus, whatever the environment does, it can't influence the *other* component of the states w described by Φ.

Precision is a technical property common to separation-style logics, though here it has a somewhat different flavor. Precision ensures that for every value g, $\Phi(g)$ precisely describes the underlying heaps of its circumscribed states; that is, each state $\Phi(g)$ is uniquely determined by its heap erasure.

D Definition of Hoare Ordering $(p_1, q_1) \sqsubseteq (p_2, q_2)$

Given preconditions p_1, p_2 and postconditions q_1, q_2, the judgment $(p_1, q_1) \sqsubseteq (p_2, q_2)$ generalizes the usual Hoare logic side conditions on the rule of consequence about strengthening the precondition $p_2 \implies p_1$ and weakening the postcondition $q_1 \implies q_2$. The generalization is required in FCSL because of the local scope of logical variable. In first order Hoare logics, the logical variables have global scope, so the above implications over p_1, p_2 and q_1, q_2 suffice. In FCSL, the logical variables have scope locally over Hoare triples, and this scope has to be reflected in the semantic definition of $\sqsubseteq$ by introducing quantifiers. The definition is similar to the one of Kleymann [9].

$$
\begin{array}{l}
(p_1, q_1) \sqsubseteq (p_2, q_2) \iff \\
\quad \forall w\, w'.\ (w \models \exists \bar{v}_2.\, p_2 \implies w \models \exists \bar{v}_1.\, p_1) \wedge \\
\qquad ((\forall \bar{v}_1\ res.\, w \models p_1 \implies w' \models q_1) \implies (\forall \bar{v}_2\ res.\, w \models p_2 \implies w' \models q_2))
\end{array}
$$

where $v_i = \mathsf{FLV}(p_i, q_i)$ are the free logical variables. The definition makes it apparent that the Hoare triple $\{p\}\, c\, \{q\}@U$ is essentially a syntactic sugar for a different kind of Hoare triple, which may be written as:

$$\{w.\, \exists \bar{v}.\, w \models p\}\, c\, \{res\ w\ w'.\, \forall \bar{v}.\, w \models p \implies w' \models q\}@U$$

where $v = \mathsf{FLV}(p, q)$. In this alternative Hoare triple, the postconditions are predicates ranging over input and output states w and w' (they are thus called binary postconditions). The advantage of the alternative Hoare triple is that the logical variables are explicitly bound, making their scoping explicit. In our Coq implementation we use this alternative formulation of Hoare triples.

E A Concurroid for the Ticketed Lock

In this section, we instantiate the presented framework by constructing from scratch a concurroid for another famous fine-grained locking algorithm—a ticketed lock [4]. Unlike the CAS-based spin lock, which we took as a running example, this algorithm, although simple, provides fairness guarantees for competing threads. We start by giving a simplified implementation of the algorithm and building the intuition about the states and transitions of the concurroid, implementing its protocol.

E.1 Reference implementation

As a reference implementation of the ticketed lock, we use the one from the work on Concurrent Abstract Predicates [4].

```
lock = {                          unlock = {
  n := TAKE();                     DROP_TKT();
  while (!TRY(n)) skip;           }
}
```

The three *atomic* actions `TAKE`, `TRY(n)` and `DROP_TKT` have the following operational meaning:

```
TAKE()     = { fetch_and_increment(next); }
TRY(n)     = { return (n == owner); }
DROP_TKT() = { fetch_and_increment(owner); }
```

As a minor difference, we assume a pointer to the lock structure being fixed and omit it from the code, referring instead to its two essential fields: `owner` and `next`. These two fields are natural numbers, describing the state of the ticketed lock system. We implemented `TAKE()` and `DROP_TKT()` as `fetch_and_increment`, which is is a standard atomic operation implemented in most of the modern architectures, operationally equivalent to the following sequence of commands:

```
fetch_and_increment(ptr) = { tmp := ptr; ptr++; return tmp; }
```

The intuition behind the protocol comes from the idea of the organization of the service in a bakery: `next` corresponds to the current ticket available at the dispenser (but not yet taken!), whereas `owner` indicates a ticket number, an owner of which is called to be served, or is currently being served. Hence, the `lock` procedure corresponds to drawing a ticket from the dispenser and a subsequent repetitive attempt to be served, whereas `unlock` corresponds to throwing the ticket away and increasing the ticket counter, so the next client could be served (if there is one in a waiting line).

E.2 States of the concurroid

The intuition for the construction of a ticketed lock concurroid can be built from the idea of *splitting* the resources in the spirit of the Subjective Concurrent Separation Logic [10]. What are the primary resources that a ticketed lock deals with? Those are tickets, of course! Some of them can be taken by a current thread, some by others. If we make an assumption that each ticket is thrown away right after its owner has been served, we will see a simple invariant: all tickets in the system are from the range $[n_1, n_2)$, where n_1 and n_2 are natural numbers, corresponding to the values of `owner` (currently being served or about to be) and `next` (next ticked to be issued), hence non-inclusion on the right side of the range.

As in the case of the CAS-based spin lock (Section 2), the *self* and *other* components of the ticketed lock concurroid are pairs of two PCMs: the PCM of *finite sets* of

natural numbers, accounting for the tickets ($\mathsf{t_S}$ and $\mathsf{t_O}$), and a generic PCM accounting in order to the heap invariant associated with this particular lock: $\mathsf{a_S}$ and $\mathsf{a_O}$. Similarly to the CAS-based spin lock, the ticketed lock is parametrized by a resource invariant I. We will denote the whole concurroid with the label tlock, pointers *own* and *nxt* and invariant I by $T_{\{\mathsf{tlock},own,nxt,I\}}$. Let us start by defining a predicate, describing valid states of the ticketed lock concurroid.

Definition 7 ($T_{\{\mathsf{tlock},own,nxt,I\}}$ **state invariant** $\mathcal{W}_T$**).**

$$\mathcal{W}_T \mathrel{\widehat{=}} \mathsf{tlock} \overset{s}{\mapsto} (\mathsf{t_S}, \mathsf{a_S}) \wedge \mathsf{tlock} \overset{o}{\mapsto} (\mathsf{t_O}, \mathsf{a_O}) \wedge \mathsf{tlock} \overset{j}{\mapsto} (own \to \langle n_1, b\rangle) \cup (nxt \to n_2) \cup h \wedge$$
$$n_1 \le n_2 \wedge (\mathsf{t_S} \bullet \mathsf{t_O}) = \{n \mid n_1 \le n < n_2\} \wedge$$
$$\begin{pmatrix} n_1 \in (\mathsf{t_S} \bullet \mathsf{t_O}) \wedge b = \mathsf{true} \wedge h = \mathsf{empty} \vee \\ \mathsf{if}\ \ n_1 = n_2\ \ \mathsf{then}\ \ b = \mathsf{false} \wedge I\ (\mathsf{a_S} \bullet \mathsf{a_O})\ h \\ \qquad\qquad \mathsf{else}\ \ n_1 \in (\mathsf{t_S} \bullet \mathsf{t_O}) \wedge b = \mathsf{false} \wedge I\ (\mathsf{a_S} \bullet \mathsf{a_O})\ h \end{pmatrix}$$

Definition 7 highlights a number of important invariants of the ticketed lock:

- The first line of the conjunction specifies the shape of the concurroid states. Since the *joint* component is always a heap, we needed to introduce a "ghost" value b, such that $own \to \langle n_1, b\rangle$, which tracks if the lock is taken or not. This is a purely logical artifact, which simplifies the reasoning and it can be safely erased in the actual implementation, making this entry look just as $(own \to n_1)$. To make it clear that b is auxiliary state, we could have introduced a separate label whose *joint* part contains only b. In our implementation, *joint* components need not contain only heaps, but can contain arbitrary non-heap types, whose values are by default considered auxiliary. Thus, the addition of a new label with a boolean contents b would suffice to declare b as an auxiliary state. However, for simplicity, we omitted such generalization.
- The second line states the invariant with respect to the distributed tickets. Indeed, the number of a ticket currently being (or just about to be) served is less than or equal to the number of the ticket in the dispenser, hence, $n_1 \le n_2$. The tickets in the queue range from n_1 to n_2, and n_2 has not yet been assigned to any thread.
- The last three lines outline the essential "states" of the system.
  - One case is the lock being locked. It corresponds to a thread with some ticket n_1 (captured either by $\mathsf{t_S}$ or $\mathsf{t_O}$) being served. Therefore, the ownership over the heap is being transferred to the client ($h = \mathsf{empty}$) and the logical flag b is indicating the locking ($b = \mathsf{true}$). We will refer to this state as *locked*.
  - Another case corresponds to the situation when there are no more threads awaiting for service or being served. This is the case $n_1 = n_2$. Therefore, the lock is not taken ($b = \mathsf{false}$) and has the ownership over the heap with the lock invariant holding over it ($I\ (\mathsf{a_S} \bullet \mathsf{a_O})\ h$). We will refer to this state as *unlocked*.
  - The least obvious case is the situation when *no* thread is being served, but one is just about to be. This is the thread having the ticket n_1 (hence $n_1 \in (\mathsf{t_S} \bullet \mathsf{t_O})$), but the lock is not yet taken ($b = \mathsf{false}$) and has the ownership over the heap ($I\ (\mathsf{a_S} \bullet \mathsf{a_O})\ h$). We will refer to such state as *transit*.

It is important to notice that the awkward addition of the logical flag b to the *joint* state is a byproduct of co-existence of "locked" and "transit" states of the concurroid and our generality with respect to the invariant I. In particular, had we assumed $I\ (\mathsf{a_S} \bullet \mathsf{a_O})\ h \implies h \neq \mathsf{empty}$, we could use this fact in order to distinguish between the "locked" and "transit" states. Finally, as it will be shown, the flag b plays an important role only when verifying the body of the `lock` procedure, and can be abstracted away for convenience of the clients of `lock` and `unlock`.

E.3 Transitions of the concurroid

In contrast with the CAS-based spin lock, whose internal transitions do not allow any components of the state to change, the internal transitions of the ticketed lock concurroid account for *taking* the tickets from the dispenser and changing the corresponding counter *nxt* in the *joint* part of the state. Again, describing the transitions on states w and w', we implicitly assume $w.o = w'.o$.

Definition 8 ($T_{\{\mathsf{tlock},own,nxt,I\}}$ **internal transitions** τ_T**).**

$$\begin{array}{rll}(w,w') \in \tau_T \iff & w.s & = \mathsf{tlock} \to (\mathsf{t_S}, \mathsf{a_S}),\ w.j = \mathsf{tlock} \to (nxt \to n_2 \cup h),\\ & w'.s & = \mathsf{tlock} \to (\mathsf{t'_S}, \mathsf{a_S}),\ w.j = \mathsf{tlock} \to (nxt \to n'_2 \cup h),\\ & n_2 \leq n'_2 & \end{array}$$

Notice that in Definition 8 we don't specify the rest of the *joint* heap h. What is important is that it remains unchanged. The *nxt* counter may only increase, and this change is reflected by changing the *self* auxiliary component $\mathsf{t_S}$ to $\mathsf{t'_S}$, assuming that the resulting state still adheres to the state predicate $\mathcal{W}_T$ (Definition 7).

Definition 9 ($T_{\{\mathsf{tlock},own,nxt,I\}}$ **external transitions,** α_T **and** ρ_T**).**

$$\begin{array}{rll}(w,w') \in \alpha_T\ h \iff & w.s & = \mathsf{tlock} \to (\mathsf{t_S} \cup \{n_1\}, \mathsf{a_S}),\\ & w.j & = \mathsf{tlock} \to (own \to \langle n_1, \mathsf{true}\rangle \cup nxt \to n_2),\\ & w'.s & = \mathsf{lock} \to (\mathsf{t_S}, \mathsf{a'_S}),\\ & w'.j & = \mathsf{tlock} \to (own \to \langle n_1 + 1, \mathsf{false}\rangle \cup nxt \to n_2 \cup h)\end{array}$$

$$\begin{array}{rll}(w,w') \in \rho_T\ h \iff & w.s & = \mathsf{lock} \to (\mathsf{t_S} \cup \{n_1\}, \mathsf{a_S}),\\ & w.j & = \mathsf{tlock} \to (own \to \langle n_1, \mathsf{false}\rangle \cup nxt \to n_2 \cup h),\\ & w'.s & = \mathsf{lock} \to (\mathsf{t_S} \cup \{n_1\}, \mathsf{a_S}),\\ & w'.j & = \mathsf{tlock} \to (own \to \langle n_1, \mathsf{true}\rangle \cup nxt \to n_2)\end{array}$$

In Definition 9, the acquire transition α_T transfers the heap h to the *joint* part of the state $w'.j$, increases the service counter, hence $n_1 + 1$, and flips the flag b from true to false. The transition is only possible when the ticket number n_1 is "registered" in the *self*-set $\mathsf{t_S}$ in the initial state. Similarly, the release transition ρ_T is only possible when the ticked being called is owned by the *self* set $\mathsf{t_S}$. In such a case, the ownership of the heap h is given away to the calling thread.

We now have all ingredients to define the concurroid for ticketed locks.

Definition 10 (The concurroid for the ticketed lock).

$T_{\{\mathsf{tlock},own,nxt,I\}} = (\{\mathsf{tlock}\}, \mathcal{W}_T, \tau_T, (\alpha_T, \rho_T))$*, where* $\mathcal{W}_T$ *is given by Definition 7 and* τ_T, α_T *and* ρ_T *are defined in Definitions 8 and 9.*

E.4 Atomic actions of a ticketed lock

We now define the three atomic actions, used in the implementation of the locking/unlocking procedures in Section E.1.

The action take corresponds to internal transitions of the concurroid $T_{\{\mathsf{tlock},own,nxt,I\}}$ and returns the number of the drawn ticket as its result. Hence the following specifications, according to the definitions of Section 5:

Definition 11 (take action). take = $(T_{\{\mathsf{tlock},own,nxt,I\}}, \mathsf{nat}, \sigma, \mu)$*, where*

$$w \in \sigma \iff w = \mathsf{tlock} \to [(\mathsf{t_S}, \mathsf{a_S}) \mid nxt \to n_2 \cup h \mid (\mathsf{t_O}, \mathsf{a_O})]$$

$$\begin{array}{lll} (w, w', res) \in \mu \iff & w = \mathsf{tlock} \to [(\mathsf{t_S}, & \mathsf{a_S}) \mid nxt \to n_2 \quad \cup h \mid (\mathsf{t_O}, \mathsf{a_O})] \wedge \\ & w' = \mathsf{tlock} \to [(\mathsf{t_S} \cup \{n_2\}, \mathsf{a_S}) \mid nxt \to (n_2 + 1) \cup h \mid (\mathsf{t_O}, \mathsf{a_O})] \wedge \\ & res = n_2 \end{array}$$

By doing *flattening*, one can see that operationally take corresponds to an atomic *fetch-and-increment* of the pointer *nxt*. Formally, the fetch-and-increment operation can be is defined in Section A. Two other actions, try and droptkt account for the heap ownership transfer and, hence, correspond to internal transitions of the entanglement $P \bowtie T_{\{\mathsf{tlock},own,nxt,I\}}$.

Definition 12 (try action). $\forall n : \mathsf{nat}.\ \mathsf{try}(n) = (P \bowtie T_{\{\mathsf{tlock},own,nxt,I\}}, \mathsf{bool}, \sigma, \mu)$*, where*

$$\begin{array}{l} w \in \sigma \iff \\ \quad w = \mathsf{priv} \to [\mathsf{empty} \mid \mathsf{empty} \mid h_\mathsf{O}] \cup \\ \qquad \mathsf{tlock} \to [(\mathsf{t_S} \cup \{n\}, \mathsf{a_S}) \mid own \to \langle n_1, b\rangle \cup nxt \to n_2 \cup h \mid (\mathsf{t_O}, \mathsf{a_O})] \end{array}$$

$$\begin{array}{lll} (w, w', res) \in \mu \iff & & \\ \quad w = \mathsf{priv} \to [\mathsf{empty} \mid \mathsf{empty} \mid h_\mathsf{O}] \cup & & \\ \qquad \mathsf{tlock} \to [(\mathsf{t_S} \cup \{n\}, \mathsf{a_S}) \mid own \to \langle n_1, b\rangle \cup nxt \to n_2 \cup h \mid (\mathsf{t_O}, \mathsf{a_O})] \wedge & & \\ \mathsf{if}\ n = n_1 & & \\ \mathsf{then} \quad w' = \mathsf{priv} \to [h \mid \mathsf{empty} \mid h_\mathsf{O}] \cup & & \\ \qquad \mathsf{tlock} \to [(\mathsf{t_S} \cup \{n\}, \mathsf{a_S}) \mid own \to \langle n_1, \mathsf{true}\rangle \cup nxt \to n_2 \mid (\mathsf{t_O}, \mathsf{a_O})] \wedge & & \\ \quad res = \mathsf{true} \ \wedge\ I\ (\mathsf{a_S} \bullet \mathsf{a_O})\ h & & \\ \mathsf{else} \quad w' = w \ \wedge\ res = \mathsf{false} & & \end{array}$$

One can see that the action try(n) corresponds to reading from the heap cell $own \to -$ (hence, it's operational), but it has a interesting effect on the auxiliary state. In particular, if the comparison succeeds (then-branch), the ownership over heap h is transferred to the *self*-part of priv, and logical flag is flipped, which corresponds to the internal transition of the entanglement $P \bowtie T_{\{\mathsf{tlock},own,nxt,I\}}$. Otherwise, the thread has approached the counter with a ticket that is not called, and thus is denied service for now. The state therefore doesn't change, which corresponds to the reflexivity of the internal transition.

The action can only be performed if the ticket n, which is being checked for the service, is present in the *self* part, hence $\mathsf{t_S} \cup \{n\}$ (that is the thread indeed has right to *try* claim the service).

Definition 13 (droptkt action). $\mathsf{droptkt} = (T_{\{\mathsf{tlock},own,nxt,I\}}, \mathsf{unit}, \sigma, \mu)$*, where*

$$
\begin{array}{lll}
w \in \sigma & \iff & w = \mathsf{priv} \to [h \mid \mathsf{empty} \mid h_\mathsf{O}] \cup \\
& & \quad \mathsf{tlock} \to [(\mathsf{t_S} \cup \{n_1\}, \mathsf{a_S}) \mid own \to \langle n_1, b\rangle \cup nxt \to n_2 \mid (\mathsf{t_O}, \mathsf{a_O})] \wedge \\
& & I\,(f(\mathsf{a_S}) \bullet \mathsf{a_O})\,h \ \wedge \ f \textit{ is local} \\
\\
(w, w', res) \in \mu & \iff & w = \mathsf{priv} \to [h \mid \mathsf{empty} \mid h_\mathsf{O}] \cup \\
& & \quad \mathsf{tlock} \to [(\mathsf{t_S} \cup \{n_1\}, \mathsf{a_S}) \mid own \to \langle n_1, b\rangle \cup nxt \to n_2 \mid (\mathsf{t_O}, \mathsf{a_O})] \wedge \\
& & I\,(f(\mathsf{a_S}) \bullet \mathsf{a_O})\,h \ \wedge \ f \textit{ is local} \wedge \\
& & w' = \mathsf{priv} \to [\mathsf{empty} \mid \mathsf{empty} \mid h_\mathsf{O}] \cup \\
& & \quad \mathsf{tlock} \to [(\mathsf{t_S}, \mathsf{a_S}) \mid own \to \langle n_1 + 1, \mathsf{false}\rangle \cup nxt \to n_2 \cup h \mid (\mathsf{t_O}, \mathsf{a_O})]
\end{array}
$$

In the definition of $\mathsf{droptkt}$, the *self*-contribution with respect to the abstract state $\mathsf{a_S}$ is captured via some *auxiliary function* f, which is required to be *local* [10]. Another important change is removing a ticket n_1 from the *self*-set $\mathsf{t_S}$, once the service is finished (*i.e.*, the lock is unlocked).

E.5 Lock predicates

In order to verify the `lock` and `unlock` procedures, we first introduce a number of auxiliary predicates to simplify the presentation of the reasoning.

Definition 14 ($T_{\{\mathsf{tlock},own,nxt,I\}}$ predicates).

$$\mathsf{isLock}_{\mathsf{tlock}}(\mathsf{a_S}) \ \widehat{=} \ \mathsf{tlock} \overset{s}{\mapsto} (\mathsf{t_S}, \mathsf{a_S})$$

$$
\begin{array}{rcl}
\mathsf{canTry}_{\mathsf{tlock}}(n, \mathsf{a_S}) & \widehat{=} & \mathsf{tlock} \overset{s}{\mapsto} (\mathsf{t_S} \cup \{n\}, \mathsf{a_S}) \wedge \mathsf{tlock} \overset{j}{\mapsto} (own \to \langle n_1, b\rangle \cup h) \wedge \\
& & (n = n_1) \implies \neg b
\end{array}
$$

$$\mathsf{Locked}_{\mathsf{tlock}}(\mathsf{a_S}, \mathsf{a_O}) \ \widehat{=} \ \mathsf{tlock} \overset{s}{\mapsto} (\mathsf{t_S} \cup \{n_1\}, \mathsf{a_S}) \wedge \mathsf{tlock} \overset{j}{\mapsto} (own \to \langle n_1, \mathsf{true}\rangle \cup h)$$

In Definition 14 and further we implicitly treat all non-bound logical variables as existentially quantified. The predicate $\mathsf{isLock}_{\mathsf{tlock}}$ identifies a state w as one satisfying to the structure of the $T_{\{\mathsf{tlock},own,nxt,I\}}$ concurroid. The predicate $\mathsf{canTry}_{\mathsf{tlock}}$ asserts that the current thread can make an attempt to acquire the lock with the ticket n, but it might fail. The auxiliary condition $(n = n_1) \implies \neg b$ is essential for verifying the body of `lock` in order to exclude the bogus situation when the same thread tries to re-acquire the lock which it already owns.

Curiously, the $\mathsf{isLock}_{\mathsf{tlock}}$ and $\mathsf{Locked}_{\mathsf{tlock}}$ predicates can be considered as implementation of the *abstract* interface for locks, as presented by the work on CAP [4]: we deliberately picked the same name for ours. In particular, we can prove the similar properties for ours predicates with respect to the locking intuition:

Lemma 3. $\mathsf{Locked}_{\mathsf{tlock}}(\mathsf{a_S}, \mathsf{a_O}) \implies \mathsf{isLock}_{\mathsf{tlock}}(\mathsf{a_S})$.

Lemma 4 (Mutual exclusion for Ticketed Lock).

$$w \models \mathsf{Locked}_{\mathsf{tlock}}(\mathsf{a}_S, \mathsf{a}_O) \ \wedge \ w^{\top} \models \mathsf{Locked}_{\mathsf{tlock}}(\mathsf{a}_O, \mathsf{a}_S) \implies \mathsf{False}.$$

Lemma 4 is a "subjective" formulation of the mutual exclusion property, stating that the lock can be simultaneously acquired by at most one thread. We address the interested reader to our formal development in Coq [11] for the proofs of these and some other lemmas about locks as well as for implementation of the same abstract interface for the CAS-based spin lock.

E.6 Verifying locking procedures

We culminate the development of this appendix with the verification of the `lock` procedure for the ticketed lock (Section E.1).[8]

$$\begin{array}{c} \left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{isLock}_{\mathsf{tlock}}(\mathsf{a}_S)\right\} \\ \mathsf{lock} \\ \left\{\exists h, \mathsf{a}_O, \mathsf{priv} \overset{s}{\mapsto} h * \mathsf{Locked}_{\mathsf{tlock}}(\mathsf{a}_S, \mathsf{a}_O) \ \wedge \ I\,(\mathsf{a}_S \bullet \mathsf{a}_O)\,h\right\} \end{array} \tag{6}$$

Since in our logic imperative `while`-loops are implemented as a fixed point iteration of a tail-recursive function, for such functions a specification should be defined by the programmer (same is true for loop invariants in the standard Hoare logic). We begin by assuming for each auxiliary element of PCM a_S and a ticket number n a family of curried functions $\mathit{loop}(n : \mathsf{nat}) : \mathsf{unit} \to \mathsf{unit}$ with the following FCSL specification:

$$\begin{array}{c} \left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{canTry}_{\mathsf{tlock}}(n, \mathsf{a}_S)\right\} \\ \mathit{loop}(n)(\_) \\ \left\{\exists h, \mathsf{a}_O, \mathsf{priv} \overset{s}{\mapsto} h * \mathsf{Locked}_{\mathsf{tlock}}(\mathsf{a}_S, \mathsf{a}_O) \ \wedge \ I\,(\mathsf{a}_S \bullet \mathsf{a}_O)\,h\right\} \end{array} \tag{7}$$

We proceed by unfolding the definition of the locking procedure and presenting the proof outline.

[8] Verification of `unlock` is significantly simpler, and we address the reader to our formal development for the details [11].

$\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{isLock}_{\mathsf{tlock}}(\mathsf{a_S})\right\}$
$n \leftarrow \mathsf{inject}'(\mathsf{take});$
$\left\{\begin{array}{l}\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{tlock} \overset{s}{\mapsto} (\mathsf{t_S} \cup \{n_2\}, \mathsf{a_S}) \wedge \mathsf{tlock} \overset{j}{\mapsto} (own \rightarrow - \cup nxt \rightarrow n_2' \cup h') \wedge \\ n = n_2 \wedge n_2 \leq n_2'\end{array}\right\}$
$\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{canTry}_{\mathsf{tlock}}(n, \mathsf{a_S})\right\}$
$\mathsf{fix}.loop(n).\ \_\ .\ ($
 $\mathsf{if}\ (\mathsf{try}(n))\ \mathsf{then}$
  $\left\{\mathsf{priv} \overset{s}{\mapsto} h * \mathsf{tlock} \rightarrow [(\mathsf{t_S}, \mathsf{a_S}) \mid own \rightarrow \langle n, \mathsf{true}\rangle \cup nxt \rightarrow n_2'' \mid (\mathsf{t_O'}, \mathsf{a_O})] \wedge I\ (\mathsf{a_S} \bullet \mathsf{a_O})\ h\right\}$
  $\mathsf{return}\ ();$
  $\left\{\mathsf{priv} \overset{s}{\mapsto} h * \mathsf{Locked}_{\mathsf{tlock}}(\mathsf{a_S}, \mathsf{a_O}) \ \wedge\ I\ (\mathsf{a_S} \bullet \mathsf{a_O})\ h\right\}$
 else
  $\left\{\mathsf{priv} \overset{s}{\mapsto} \mathsf{empty} * \mathsf{canTry}_{\mathsf{tlock}}(n, \mathsf{a_S})\right\}$
  $loop(n)(\_);$
  $\left\{\exists h, \mathsf{a_O}, \mathsf{priv} \overset{s}{\mapsto} h * \mathsf{Locked}_{\mathsf{tlock}}(\mathsf{a_S}, \mathsf{a_O}) \ \wedge\ I\ (\mathsf{a_S} \bullet \mathsf{a_O})\ h\right\}$
$)(\_)$
$\left\{\exists h, \mathsf{a_O}, \mathsf{priv} \overset{s}{\mapsto} h * \mathsf{Locked}_{\mathsf{tlock}}(\mathsf{a_S}, \mathsf{a_O}) \ \wedge\ I\ (\mathsf{a_S} \bullet \mathsf{a_O})\ h\right\}$

In fact, the proof has been implicitly split into two parts: (a) proving the correctness of the *loop* specification (7) in the body of the locking procedure by the rule FIX (Figure 2) and (b) proving the specification of the locking itself, assuming the *loop* spec. In the outline we have blended them both together, as the proof of (7) is straightforward: just by reading the specification of the try action, we know that in then-branch the heap h has been transferred to *self*-part of priv. Moreover, in the proof outline we omitted some reasoning about stability, which, indeed, has been taken into account. In particular, in the assertion right after the line "if (try(n)) then", we mention n_2'' rather than n_2', since this counter could have been changed by the *other* threads. These subtle details become clear in the mechanized proofs and are omitted above for the sake of brevity.

Finally, we have employed a slightly different version of injection, which we emphasized by using the inject′ command. It differs from one we described in Section 3 by the form of entanglement it handles. The original rule INJECT enabled concurroid framing with respect to the right operand of $\bowtie$. In contrast, the following rule enables concurroid framing with respect to the left operand of $\bowtie$, or right operand of the "reversed" operator $\ltimes$:

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

The rule was, indeed, proved sound as well [11]. In general, we have actually proved the injection rule sound for any operator satisfying a number of general properties, and then shown that $\bowtie$ and $\ltimes$ (as well as some other combinators), satisfy these general properties.

F Semantics and Soundness

The semantic model for FCSL programs largely relies on the denotational semantic of *action trees* [10], which implement finite, partial approximations of the behavior of FCSL commands. Action trees are a generalization of the Brookes' action traces 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 the following: 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 introduce a number of notions in order to formalize the safety of FCSL.

The high-level view of the development is as follows. We first define semantic behavior (*i.e.*, operational semantics) for action trees wrt. a program state (Section F.1). We consider program state that, in addition to concrete heaps includes the logical information such as auxiliary state, or division of state between concurroid labels. Thus, the operational semantics of trees is *instrumented*. In Section F.4, we prove the standard erasure result of shared-memory concurrency logics, whereby logical information in program states doesn't influence the result of the computation.

Taking the low-level operational semantics of threes from Section F.1 as a base for our soundness result, we relate it to the high-level transitions of a concurroid by an always predicate (Section F.5) that ensures a tree is resilient to any amount of a concurroid's rely-interference, and that all operational steps by a tree are *safe* (*i.e.*, providing an analogue to *preservation* property for the *action safety* predicate) and correspond to concurroid interference.

The denotational semantics (Section F.6) interprets Hoare tuple judgments by the *monadic Hoare type* $\{\!|p|\!\}A\{\!|q|\!\}@U$, which is a *complete lattice* of trees that are always-safe to run from any initial configuration that satisfies precondition p and if they terminate produce a final configuration that satisfies postcondition q (under possible interference from programs respecting the transitions of the concurroid U). The complete lattice structure makes the semantic domain suitable for modeling recursion. A command is denoted by the set of its tree approximations, and a procedure is denoted by a function into a set of trees. The *soundness* of FCSL (Section F.7) follows from showing that the denotations of the commands listed in Section 5 satisfy the appropriate always predicate (*i.e.*, adhere to the *progress-and-preservation* property), and that always satisfies certain closure conditions.

We choose the Calculus of Inductive Constructions (CiC) [1, 16] as our meta logic. This has several important benefits. First, we can define a *shallow embedding* of FCSL into CiC that allows us to program and prove directly *with the semantic objects*, thus immediately lifting FCSL 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 metatheory, including the definition of action trees by *iterated inductive definitions* [16], specification-level functions, and programming-level higher-order procedures. Finally, we were able to mechanize the entire semantics and metatheory [11] in the Coq proof assistant implementation of CiC.

Following [10], we use set-theoretic and type-theoretic notation as appropriate. The reader unconcerned with the fine points of the type theory, may read a typing judgment $x : A$ as a set membership predicate $x \in A$.

F.1 Action trees and their operational semantics

The type family tree $U\ A$ of A-returning action trees is defined by an iterated inductive definition, as follows.

Definition 15 (Action trees).

Unfinished	: tree $U\ A$
Ret $(v : A)$	: tree $U\ A$
Act $(a : (U, A, \sigma, \mu))$	: tree $U\ A$
Seq $(t :$ tree $U\ B)$ $(k : B \to$ tree $U\ A)$	: tree $U\ A$
Par $(t_1 :$ tree $U\ B_1)$ $(t_2 :$ tree $U\ B_2)$ $(k : B_1 \times B_2 \to$ tree $U\ A)$	: tree $U\ A$
Inject $(t :$ tree $V\ A)$	: tree $U\ A$
Hide$_{\widehat{\Phi},g}$ $(t :$ tree $V\ A)$	: tree $U\ A$

Most of the constructors in Definition 15 are self-explanatory. Since trees have finite depth, they can only *approximate* divergent 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 (U, A, σ, μ), defined in concurroid U (Section 5). 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 $\to$ includes case-analysis, the continuation may branch upon the argument, which captures the pure computation of conditionals. This closely corresponds to the operational intuition and leads to a straightforward denotational semantics.

The Inject constructor embeds a tree $t :$ tree $V\ A$ (of a *different*, *i.e.*, smaller, concurroid V) for the underlying computation and generates a tree in the concurroid U. V and U are not arbitrary, but it has to be possible to *inject* V into U, as we shall define shortly. In particular, we will be able to prove that we can inject V into $U = V \rtimes W$ for any W.

The Hide constructor embeds a *generalized refinement function* $\widehat{\Phi}$, an abstract state g and a tree $t :$ tree $V\ A$ for the underlying computation and generates a tree in the concurroid U. Again, U and V are not arbitrary but it has to be possible to *refine* V into U. In particular, we will be able to prove the we can refine $U = P \rtimes V_1 \rtimes V_2$ into $V = P \rtimes V_1$.

In general, in Coq, the proof that V injects into U, or that V refines U, has to be an annotation on the Inject and Hide constructors, but we omit the annotations for brevity.

F.2 Injection and Refinement

We define the predicate injects $V\ U$, which intuitively means that the "larger" concurroid U can be considered as an entanglement of the "smaller" concurroid V with some additional concurroid W.

Definition 16 (Injection predicate).
injects $V\ U \iff$ *there exists W, such that the following three statements hold:*

1. $w \in \mathcal{W}_U \iff w = w_1 \cup w_2 \wedge w_1 \in \mathcal{W}_V \wedge w_2 \in \mathcal{W}_W$;
2. $\forall w_1\ w_2\ w.\ w_1 \cup w \in \mathcal{W}_U \wedge w_2 \cup w \in \mathcal{W}_U \wedge (w_1, w_2) \in \tau_V \implies (w_1 \cup w, w_2 \cup w) \in \tau_U$;
3. $\forall w_1\ w_1'\ w_2\ w_2'.$

$$s_1 \in \mathcal{W}_V \wedge s_2 \in \mathcal{W}_V \wedge (s_1 \cup s_1', s_2 \cup s_2') \in \mathcal{G}_U \implies (s_1, s_2) \in \mathcal{G}_V \wedge (s_1', s_2') \in \mathcal{G}_W, \quad \textit{where}$$

$\mathcal{G}_X$ *is a* guarantee *relation of a concurroid X (Definition 4).*

The first requirement ensures that the states of U can be constructed as Cartesian products of states of V and W. The second requirement ensures that internal transitions of the smaller concurroid V do not break the coherence of it enclosing concurroid U. The third requirement ensures that the transitions within U can be always uniquely split to the transitions done by V and by the "addition" W. Notably, the first requirement ensures that if injects $V\ U$ holds, there always exists a "smaller" state w_V, such that $w_V \in \mathcal{W}_V$ for any $w_U \in \mathcal{W}_U$.

We will also introduce another auxiliary definition in order to have a handle to the existential concurroid W in Definition 16.

Definition 17. *We will write* $U = V \bowtie W$, *when* injects $V\ U$ *and W is an additional concurroid from Definition 16.*

The notation $\bowtie$ hints that already familiar entanglement operators $\rtimes$ and $\ltimes$ can be seen as particular cases of $\bowtie$.

Lemma 5. injects $V\ (V \rtimes W)$ *for every V and W with disjoint label sets.*

We next define the refinement relation on concurroids. The concurroid U is refined into V, if we can, intuitively, elaborate the states of U into those of V. In other words, if V's states can be seen to contain the states of U, plus some other additional state. Abstractly, we capture the dependence by positing an elaboration predicate $\widehat{\Phi}$ between the state spaces of U and V, with a number of properties shown below. Additionally, $\widehat{\Phi}$ takes a value $g : G$ of user-specified type, which is an abstraction of the mentioned additional state. Thus, we read $\widehat{\Phi}(g)(w, w')$ to say that g elaborates w into w' (via $\widehat{\Phi}$).

Definition 18 (Elaboration predicate).
Given a type G of abstract states and a function $\widehat{\Phi} : G \to (\mathcal{W}_U, \mathcal{W}_V) \to$ prop,
refines $U\ V\ \widehat{\Phi} \iff$ *the following statements hold:*

1. $\widehat{\Phi}(g)(w, w_1) \wedge \widehat{\Phi}(g)(w, w_2) \implies w_1 = w_2$;

2. $\widehat{\Phi}(g_1)(w_1, w) \land \widehat{\Phi}(g_1)(w_2, w) \implies g_1 = g_2 \land w_1 = w_2$;
3. $\widehat{\Phi}(g)(w, w') \implies w \in \mathcal{W}_U \land w' \in \mathcal{W}_V$;
4. $\widehat{\Phi}(g)(w_1, w_1') \land (w_1, w_2) \in \mathcal{R}_U \implies \exists w_2', \widehat{\Phi}(g)(w_2, w_2') \land (w_1', w_2') \in \mathcal{R}_V$;
5. $\widehat{\Phi}(g_1)(w_1, w_1') \land (w_1', w_2') \in \tau_V \implies \exists g_2\ w_2.\ \widehat{\Phi}(g_2)(w_2, w_2') \land (w_1, w_2) \in \tau_U$;
6. $\widehat{\Phi}(g_1)(w_1, w_1') \land \widehat{\Phi}(g_2)(w_2, w_2') \implies (w_1.o = w_1.o \iff w_1'.o = w_2'.o)$;
7. $\forall p. \exists q.$

$$\begin{array}{l} \forall g\ w\ w'.\ \widehat{\Phi}(g)(w \triangleright p, w') \implies \left(\exists w_1', w' = w_1' \triangleright q \land \widehat{\Phi}(g)(w \triangleleft p, w_1' \triangleleft q)\right) \quad \land \\ \forall g\ w\ w'.\ \widehat{\Phi}(g)(w, w' \triangleleft q) \implies \left(\exists w_1, w = w_1 \triangleleft p \land \widehat{\Phi}(g)(w_1 \triangleright p, w' \triangleright q)\right); \end{array}$$

8. $\widehat{\Phi}(g)(w, w') \implies \lfloor w \rfloor = \lfloor w' \rfloor$;

If the properties in Definition 18 are satisfied, we call $\widehat{\Phi}$ and elaboration predicate for U and V. In Definition 18, (1) states that $\widehat{\Phi}$ uniquely determines refined states (however, some states in U need not refine into anything in V, *i.e.*, $\widehat{\Phi}$ is a partial function); (2) states that the $\widehat{\Phi}$ is injective: abstracting back from a refinement is unique. Thus, (1) and (2) state that $\widehat{\Phi}$ is a partial bijection. (3) ensures that the elaboration maps well-formed states to well-formed states; (4) specifies that environment steps in the "coarse" world do not change introduced abstractions in the "fine" world, as environment steps in the coarse world can't see the refinement; (5) states that "abstracting back" preserves internal transitions. The properties (4) and (5) can be seen as stating commutativity of the apparent categorical diagrams. They may also be seen as stating simulation properties. In (4), a rely transitions of U can always be matched by a rely transition of V. In (5), an internal transition of V may always be matched by an internal transition of U. (6) postulates that refinement realigns *self* and *joint* parts, but doesn't move things into/out of *other*; (7) is a version of framing, namely, every every extension to *other* of a coarse state is is uniquely refined into an addition to fine state; finally, (8) ensures that refinement only deals with abstract parts of the state and does not change the heap.

Lemma 6 (Hiding and refinement).

Let U and V be concurroids, and let Φ be an abstraction function from a HIDE*; that is, Φ satisfies the properties about concurroid V from Section C. Let $\widehat{\Phi}$ be constructed as follows.*

$$\begin{array}{rl} \widehat{\Phi}(g)(w, w') \iff \exists h\ w_1\ w_2,\ w & = (\mathsf{priv} \to (h \cup \lfloor w_1 \rfloor)) \cup w_2 \land \\ w' & = (\mathsf{priv} \to h) \cup w_1 \cup w_2 \land \\ w_1 & \in \Phi(g) \land \\ w & \in \mathcal{W}_{P \rtimes U} \land w' \in \mathcal{W}_{P \rtimes U \rtimes V} \end{array}$$

Then $\widehat{\Phi}$ is a valid elaboration predicate, that is refines *$(P \rtimes U)$ $(P \rtimes U \rtimes V)$ $\widehat{\Phi}$.*

F.3 Operational semantics of action trees

The judgment for small-step operational semantics of action trees has the form $w; t \overset{\pi}{\Rightarrow}_{\mathrm{t}} w', t'$ (Figure 3). It operates on program states w and paths π to step the tree t from initial state

Fig. 3 Tree stepping $w; t \overset{\pi}{\Rightarrow}_{\mathsf{t}} w'; t'$ with respect to the path π

$$\frac{(w, w', v) \in \mu}{w; \mathsf{Act}\ (U, A, \sigma, \mu) \xrightarrow{\mathsf{ChoiceAct}}_{\mathsf{t}} w'; \mathsf{Ret}\ v} \qquad \frac{}{w; \mathsf{Par}\ (\mathsf{Ret}\ v_1)\ (\mathsf{Ret}\ v_2)\ k \xrightarrow{\mathsf{ParRet}}_{\mathsf{t}} w; k\ (v_1, v_2)}$$

$$\frac{w; t_1 \overset{\pi}{\Rightarrow}_{\mathsf{t}} w'; t_1'}{w; \mathsf{Par}\ t_1\ t_2\ k \xrightarrow{\mathsf{ParL}\ \pi}_{\mathsf{t}} w'; \mathsf{Par}\ t_1'\ t_2\ k} \qquad \frac{w; t_2 \overset{\pi}{\Rightarrow}_{\mathsf{t}} w'; t_2'}{w; \mathsf{Par}\ t_1\ t_2\ k \xrightarrow{\mathsf{ParR}\ \pi}_{\mathsf{t}} w'; \mathsf{Par}\ t_1\ t_2'\ k}$$

$$\frac{}{w; \mathsf{Seq}\ (\mathsf{Ret}\ v)\ k \xrightarrow{\mathsf{SeqRet}}_{\mathsf{t}} w; (k\ v)} \qquad \frac{w; t_1 \overset{\pi}{\Rightarrow}_{\mathsf{t}} w'; t_2}{w; \mathsf{Seq}\ t_1\ k \xrightarrow{\mathsf{SeqStep}\ \pi}_{\mathsf{t}} w'; \mathsf{Seq}\ t_2\ k}$$

$$\frac{}{w; \mathsf{Inject}\ (\mathsf{Ret}\ v) \xrightarrow{\mathsf{InjRet}}_{\mathsf{t}} w; \mathsf{Ret}\ v}$$

$$\frac{\mathsf{injects}\ V\ U \quad w_U = w_V \uplus w_0 \quad w_V; t_1 \overset{\pi}{\Rightarrow}_{\mathsf{t}} w_V'; t_2 \quad w_U' = w_V' \uplus w_0}{w_U; \mathsf{Inject}\ t_1 \xrightarrow{\mathsf{InjStep}\ \pi}_{\mathsf{t}} w_U'; \mathsf{Inject}\ t_2}$$

$$\frac{}{w; \mathsf{Hide}\ \widehat{\Phi}\ g\ (\mathsf{Ret}\ v) \xrightarrow{\mathsf{HideRet}}_{\mathsf{t}} w; \mathsf{Ret}\ v}$$

$$\frac{\mathsf{hides}\ U\ V\ \widehat{\Phi} \quad \widehat{\Phi}(g)(w_U, w_V) \quad w_V; t_1 \overset{\pi}{\Rightarrow}_{\mathsf{t}} w_V'; t_2 \quad \widehat{\Phi}(g')(w_U', w_V')}{w_U; \mathsf{Hide}\ \widehat{\Phi}\ g\ t_1 \xrightarrow{\mathsf{HideStep}\ \pi}_{\mathsf{t}} w_U'; \mathsf{Hide}\ \widehat{\Phi}\ g'\ t_2}$$

w to a reduced tree t' in ending state w'. Intuitively, the path π identifies the position in the tree to be reduced, which may be an action, or a beta redex.

$$\begin{array}{rcllllll} \pi & ::= & \mathsf{ChoiceAct} & | & \mathsf{SeqRet} & | & \mathsf{SeqStep}\ \pi & | \\ & & \mathsf{ParRet} & | & \mathsf{ParL}\ \pi & | & \mathsf{ParR}\ \pi & | \\ & & \mathsf{HideStep}\ \pi & | & \mathsf{HideRet} & | & \mathsf{InjStep}\ \pi & | \quad \mathsf{InjRet}. \end{array} \tag{8}$$

Stepping is undefined for the Unfinished and Ret trees. For the Seq and Par trees, the path π selects a β-redex and performs the appropriate reduction. For the Inject tree in the non-Ret case the reduction of the tree t_1 is performed for a smaller state w_V instead of the large state w_U, and the resulted state w_V' is "plugged" back using the addition w_0 in order to obtain the resulting large state w_U'. The situation for Hide is symmetric: the coarse-grained state w_U and the fine-grained state w_V are related by the elaboration $\widehat{\Phi}$. Note that the modifications of the changes to the abstract values g are recorded into g'.

The decorated semantics in Figure 3 may not make a step for two different reasons. The first, benign, reason is that the the chosen path π doesn't actually determine an action or a redex in the tree t. For example, we may have $t = \mathsf{Unfinished}$ and $\pi = \mathsf{ParR}$. But we can choose the right side of a parallel composition only in a tree whose head

constructor is Par, which is not the case with Unfinished. We consider such paths that don't determine an action or a redex in a tree 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, what is some of the rules can't be applied because their side-conditions are not satisfies by the current state.

We next state a form of the progress property, showing that the latter case can't occur in trees that approximate well-proved programs. We first formally define paths which are well-formed wrt. a tree.

Definition 19 (Good paths). *Let* t : tree $U\ A$ *and* π *be a path. Then the predicate* good $t\ \pi$ *is defined as follows:*

good (Act (U, A, σ, μ))	ChoiceAct	$\widehat{=}$	true
good (Seq (Ret v) _)	SeqRet	$\widehat{=}$	true
good (Seq t _)	SeqRet π	$\widehat{=}$	good $t\ \pi$
good (Par (Ret _) (Ret _) _)	ParRet	$\widehat{=}$	true
good (Par $t_1\ t_2$ _)	ParL π	$\widehat{=}$	good $t_1\ \pi$
good (Par $t_1\ t_2$ _)	ParR π	$\widehat{=}$	good $t_2\ \pi$
good (Inject (Ret _))	InjRet	$\widehat{=}$	true
good (Inject t)	InjStep π	$\widehat{=}$	good $t\ \pi$
good (Hide _ _ (Ret _))	HideRet	$\widehat{=}$	true
good (Hide _ _ t)	HideStep π	$\widehat{=}$	good $t\ \pi$
good t	π	$\widehat{=}$	false *otherwise*

We next require a notion of safety for a tree and a path.

Definition 20 (Safe path for an action tree). *Let* t : tree $U\ A$, π *be a path* π, *and state* $w \in \mathcal{W}_U$. safe $t\ \pi\ w$ *is defined as follows:*

safe (Act (U, A, σ, μ))	ChoiceAct	w	$\widehat{=}$	$(w \in \sigma)$
safe (Seq (Ret v) _)	SeqRet	w	$\widehat{=}$	true
safe (Seq t _)	SeqStep π	w	$\widehat{=}$	safe $t\ \pi\ w$
safe (Par (Ret _) (Ret _) _)	ParRet	w	$\widehat{=}$	true
safe (Par $t_1\ t_2$ _)	ParL π	w	$\widehat{=}$	safe $t_1\ \pi\ w$
safe (Par $t_1\ t_2$ _)	ParR π	w	$\widehat{=}$	safe $t_2\ \pi\ w$
safe (Inject (Ret _))	InjRet	w	$\widehat{=}$	true
safe (Inject (t : tree $V\ A$))	InjRet π	w	$\widehat{=}$	injects $V\ U \wedge w = w_V \cup w_0 \wedge$ safe $t\ \pi\ w_V$
safe (Hide _ _ (Ret _))	HideRet	w	$\widehat{=}$	true
safe (Hide $\Phi\ g$ (t : tree $V\ A$))	HideStep π	w	$\widehat{=}$	hides $U\ V\ \Phi \wedge \Phi(g)(w, w_V) \wedge$ safe $t\ \pi\ w_V$
safe t	π	w	$\widehat{=}$	true *otherwise*

The definition traverses the path and the tree, and in those cases when the tree and the path match, collects the side-conditions on the *initial states* of the operational semantics rules.

Lemma 7 (Progress on action trees). *For any concurroid U, state $w \in \mathcal{W}_U$, action tree t : tree U A and a path π, if* safe $t\ \pi\ w$*, then either of the following holds*

1. $\neg$ good $t\ \pi$*;*
2. $\exists w'\ t',\ w; t \overset{\pi}{\Rightarrow}_{\mathsf{t}} w'; t'$.

The lemma shows that if we can match the tree t with a path π and thus manage to pick an action or a redex (*i.e.*, good $t\ \pi$), then the safety predicate implies the side conditions of the inference rule determined by π.

The import of the lemma is in allowing that safety and stepping relations may be defined independently of each other. The difficulty of the proof consists in showing that the various conditions from the reduction rules, that have *not* been incorporated into the safety predicate, are nevertheless satisfied. For example, the reduction rules for Inject and Hide contain such side conditions on the *ending* states of their premisses. We have proved that even though these conditions are not incorporated by the safe predicate, they are implied by it. The proof relies on the general properties of injection and refinement listed in Section F.2.

Defining safety independently from stepping will enable us to develop a semantics in the style of Milner and Wright and Felleisen, whereby we only give meaning to programs that are well-proved in FCSL, analogously to how Milner and Wright-Felleisen semantics give meaning to only well-typed programs. As the type of the program, we will use its Hoare-tuple specification, and the denotation of a program wrt. a specification with precondition p, will only contain approximating action trees whose safety predicate is implied by p. The fact that the safety predicate actually means safety, *i.e.*, it allows stepping, is a consequence of the Progress lemma above.

F.4 Erased FCSL action trees

We next define an erased version of action trees that remove the various logical annotations. One example of such erased logical information is the proofs of injection and refinement that we implicitly elided from the INJECT and HIDE constructors. Another example is the logical information from the actions embedded into action trees, which, as we have shown, operate on *decorated* states.

We then prove that such logical information doesn't influence the stepping of the tree. In other words, states that erase to equal concrete heaps, produce output states that also erase to equal concrete heaps.

Definition 21 (Tree erasure).

Given a tree t : tree U A, its erasure $\lfloor t \rfloor$: etree A is defined as follows:

$$
\begin{array}{ll}
\lfloor \mathsf{Unfinished} \rfloor & \widehat{=}\ \mathsf{Unfinished} \\
\lfloor \mathsf{Ret}\ v \rfloor & \widehat{=}\ \mathsf{Ret}\ v \\
\lfloor \mathsf{Act}\ (U, A, \sigma, \mu) \rfloor & \widehat{=}\ \mathsf{Act}\ (A, \lfloor \sigma \rfloor, \lfloor \mu \rfloor) \\
\lfloor \mathsf{Seq}\ t\ k \rfloor & \widehat{=}\ \mathsf{Seq}\ \lfloor t \rfloor\ (\lambda v. \lfloor k\ v \rfloor) \\
\lfloor \mathsf{Par}\ t_1\ t_2\ k \rfloor & \widehat{=}\ \mathsf{Par}\ \lfloor t_1 \rfloor\ \lfloor t_1 \rfloor\ (\lambda v. \lfloor k\ v \rfloor) \\
\lfloor \mathsf{Inject}\ t \rfloor & \widehat{=}\ \mathsf{Inject}\ \lfloor t \rfloor \\
\lfloor \mathsf{Hide}\ \_\ \_\ t \rfloor & \widehat{=}\ \mathsf{Hide}\ \lfloor t \rfloor
\end{array}
$$

Theorem 2 (Erasure irrelevance). *For a concurroid U, states $w_1, w_2 \in \mathcal{W}_U$ and action trees $t_1, t_2 : \mathsf{tree}\ U\ A$, if $\lfloor w_1 \rfloor = \lfloor w_2 \rfloor$, $\lfloor t_1 \rfloor = \lfloor t_2 \rfloor$, $w_1; t_1 \overset{\pi}{\Rightarrow}_{\mathrm{t}} w_1'; t_1'$ and $w_2; t_2 \overset{\pi}{\Rightarrow}_{\mathrm{t}} w_2'; t_2'$, then $\lfloor w_1' \rfloor = \lfloor w_2' \rfloor$ and $\lfloor t_1' \rfloor = \lfloor t_2' \rfloor$.*

Ideally, we will need to prove a stronger result, stating that if the tree and state evaluate under semantics from Figure 3, so do their erased counterparts under some sort of erased semantics. However, we have not carried out this exercise due to time constraints and decided to leave it for the future work.

F.5 Auxiliary modal predicates

In this section we next define a number of *modal* predicates over *all* possible steps of execution to relate the operational semantics over heaps and concurroid transitions over admissible states.

Definition 22 (Modal predicates). *The predicates* $\mathsf{always}^{\zeta}_{U}\ w\ t\ P$, $\mathsf{always}_{U}\ w\ t\ P$ *and* $\mathsf{after}_{U}\ w\ t\ Q$ *are defined relative to a schedule ζ, concurroid U, state w, A-returning rich tree t : $\mathsf{tree}\ U\ A$ and arbitrary predicates $P : \mathsf{state} \to \mathsf{tree}\ U\ A \to \mathsf{prop}$ and $Q : A \to \mathsf{state} \to \mathsf{prop}$:*

$$
\begin{aligned}
\mathsf{always}^{\zeta}_{U}\ w\ t\ P \ \widehat{=}\ & \mathsf{if}\ \zeta = \pi :: \zeta' \\
& \mathsf{then}\ \forall w_2.(w, w_2) \in \mathcal{R}_U \implies \\
& \qquad \mathsf{safe}\ t\ \pi\ w_2 \ \wedge \\
& \qquad \left(\forall w_3\ t_2.\ w_2; t \overset{\pi}{\Rightarrow}_{\mathrm{t}} w_3; t_2 \implies \mathsf{always}^{\zeta'}_{U}\ w_3\ t_2\ P\right) \\
& \mathsf{else}\ \forall w_2.(w, w_2) \in \mathcal{R}_U \implies P\ w_2\ t
\end{aligned}
$$

$$
\mathsf{always}_{U}\ w\ t\ P \ \widehat{=}\ \forall \zeta.\mathsf{always}^{\zeta}_{U}\ w\ t\ P
$$

$$
\mathsf{after}_{U}\ w\ t\ Q \ \widehat{=}\ \mathsf{always}_{U}\ w\ t\ (\lambda\ w'\ t.\ \forall v'.(t' = \mathsf{Ret}\ v') \implies Q\ v'\ w')
$$

The predicate $\mathsf{always}_{U}\ w\ t\ P$ expresses the fact that starting from the state w, the tree t remains memory-safe and the user-chosen predicate P holds of all intermediate configurations and trees, for any schedule ζ and under any environment of the concurroid U. The helper predicate $\mathsf{always}^{\zeta}_{U}\ w\ t\ P$ is defined by induction on ζ: the concurroid U is allowed to make arbitrary rely-transitions from w to w_2, in the resulting configuration the predicate $P\ w_2\ t$ holds; moreover, if the schedule is $\pi::\zeta'$, then the resulting configuration must have a state that is safe for t, π and w_2 (so, according to Theorem 7, it can step), and, if t steps to w_3 and t_2, then the predicate recurses on ζ', w_3 and t_2. One can notice that if the predicate $\mathsf{always}^{\zeta}_{U}\ w\ t\ P$, it automatically implies "preservation" of the safe predicate at each step of reduction of the tree t.

The predicate $\mathsf{after}_{U}\ w\ t\ Q$ encodes that t is safe; however, $Q\ v'\ w'$ only holds if t steps completely to $\mathsf{Ret}\ v'$ in configuration w'.

Fig. 4 Denotational semantics of FCSL judgments.

$$
\begin{array}{ll}
[\![\cdot]\!] & \widehat{=}\ . \\
[\![\Gamma, x : A]\!] & \widehat{=}\ [\![\Gamma]\!], x : A \\
[\![\Gamma, \forall x : B.\, \{p\}\, f(x) : A\, \{q\}@U]\!] & \widehat{=}\ [\![\Gamma]\!], f : \forall x : B.\, \{\!|p|\!\}A\{\!|q|\!\}@U \\
 & \\
[\![\Gamma \vdash \{p\}\, c\ :\ A\, \{q\}@U]\!] & \widehat{=}\ [\![\Gamma]\!] \vdash_{\mathsf{CiC}} [\![c]\!] : \{\!|p|\!\}A\{\!|q|\!\}@U \\
[\![\Gamma \vdash \forall x : B.\{p\}F(x) : A\{q\}@U]\!] & \widehat{=}\ [\![\Gamma]\!] \vdash_{\mathsf{CiC}} [\![F]\!] : \forall x : B.\, \{\!|p|\!\}A\{\!|q|\!\}@U \\
[\![\Gamma \vdash e : A]\!] & \widehat{=}\ [\![\Gamma]\!] \vdash_{\mathsf{CiC}} e : A \\
[\![\Gamma \vdash (p_1, q_1) \sqsubseteq (p_2, q_2)]\!] & \widehat{=} \\
\quad [\![\Gamma]\!] \vdash_{\mathsf{CiC}} \forall w\, w'.\ (w \models \exists \bar{v}_2.\, p_2 \implies w \models \exists \bar{v}_1.\, p_1) \wedge & \\
\qquad ((\forall \bar{v}_1\ res.\, w \models p_1 \implies w' \models q_1) \implies (\forall \bar{v}_2\ res.\, w \models p_2 \implies w' \models q_2)) & \\
\quad \text{where } \bar{v}_i \ \widehat{=}\ \mathsf{FLV}(p_i, q_i) &
\end{array}
$$

F.6 Denotational semantics of FCSL

In this section, we define the denotational semantics of FCSL as a shallow embedding into CiC. For the sake of simplicity, one can think of the embedding as of a macro-expansion, which compositionally replaces FCSL programs by expressions and values of CiC.

The semantics of commands and judgments are defined simultaneously. The mutual recursion is necessary because the denotation of judgments depends on the denotation of commands and procedures, while the denotation of a fixed point procedure depends on the denotation of its procedure triple to determine the lattice in which to take the fixed point.

We denote FCSL programs as *sets* T of trees of increasing precision including the Unfinished tree, which is the coarsest possible approximation of any program:

$$\mathsf{prog}\ U\ A\ \widehat{=}\ \{T \subseteq \mathcal{P}(\mathsf{tree}\ U\ A) \mid \mathsf{Unfinished} \in T\}.$$

To model recursion, we construct a complete lattice of Hoare types to get fixed points. We use the after predicate (Definition 22) to ensure the tree approximations are memory safe, respect mutual exclusion, and satisfy their FCSL specifications.

Definition 23 (FCSL Hoare types). *For a concurroid U and type A, fix a precondition $p : \mathsf{state} \to \mathsf{prop}$ and a postcondition $q : A \to \mathsf{state} \to \mathsf{prop}$, with free logical variables $\mathsf{FLV}(p, q)$. The FCSL Hoare type $\{\!|p|\!\}A\{\!|q|\!\}@U$ is defined as follows:*

$$\{\!|p|\!\}A\{\!|q|\!\}@U\ \widehat{=}\ \left\{ T \in \mathsf{prog}\ U\ A \;\middle|\; \begin{array}{l} \forall \mathsf{FLV}(p, q)\ w\ (t \in T). \\ w \models p \wedge w \in \mathcal{W}_U \implies \mathsf{after}_U\ w\ t\ q \end{array} \right\}$$

Intuitively, the denotation of a FCSL judgment $\{p\}\, c : A\, \{q\}@U$ is the set of trees T denoting the command c, together with a proof that for any initial configuration w that satisfies the precondition p, then after executing any tree $t \in T$ from c produces some result value and final configuration that satisfy postcondition q. The definition quantifies

Fig. 5 Denotational semantics of FCSL commands.

$$
\begin{aligned}
[\![\mathsf{return}]\!]\ (v : A) &\mathrel{\widehat{=}} \{\mathsf{Unfinished}, \mathsf{Ret}\ v\} \\
[\![x \leftarrow c_1; c_2]\!] &\mathrel{\widehat{=}} \{\mathsf{Unfinished}\} \cup \{\mathsf{Seq}\ t_1\ k \mid t_1 \in [\![c_1]\!], \forall x.\, k\ x \in [\![c_2]\!]\} \\
[\![c_1 \parallel c_2]\!] &\mathrel{\widehat{=}} \{\mathsf{Unfinished}\} \cup \{\mathsf{Par}\ t_1\ t_2\ (\lambda v.\mathsf{Ret}\ v) \mid t_1 \in [\![c_1]\!], t_2 \in [\![c_2]\!]\} \\
[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] &\mathrel{\widehat{=}} \mathsf{if}\ e\ \mathsf{then}\ [\![c_1]\!]\ \mathsf{else}\ [\![c_2]\!] \\
[\![\mathsf{act}\ a]\!] &\mathrel{\widehat{=}} \{\mathsf{Unfinished}, \mathsf{Act}\ a\} \\
[\![\mathsf{inject}\ c]\!] &\mathrel{\widehat{=}} \{\mathsf{Unfinished}\} \cup \{\mathsf{Inject}\ t \mid t \in [\![c]\!]\} \\
[\![\mathsf{hide}_{\Phi,g}\ c]\!] &\mathrel{\widehat{=}} \{\mathsf{Unfinished}\} \cup \left\{\mathsf{Hide}\ \widehat{\Phi}\ g\ t \mid t \in [\![c]\!]\right\},\ \text{where} \\
&\quad\ \widehat{\Phi}\ \text{is constructed from}\ \Phi\ \text{as in Lemma 6} \\
[\![F\ (e)]\!] &\mathrel{\widehat{=}} [\![F]\!]\ (e) \\
[\![\mathsf{fix}\ f^{B,A,p,q}.\, x : A.\, c]\!] &\mathrel{\widehat{=}} \mathsf{lfp}_{\forall x:B.\, \{\!|p|\!\}A\{\!|q|\!\}@U}\ (\lambda f.\lambda x.\, [\![c]\!])
\end{aligned}
$$

over the free logical variables of p and q in order to give these variables local scope, as stipulated in Section 5. FCSL assertions (Section 3) are arbitrary CiC predicates of type $\mathsf{state} \to \mathsf{prop}$; and we model the relation $w \models p$ of FCSL as the application $p\ w$ in CiC.

Lemma 8. *The type $\{\!|p|\!\}A\{\!|q|\!\}@U$ is a complete lattice, with set union as the join operator, and* {Unfinished} *as the unit element.*

The type $\forall x : B.\, \{\!|p|\!\}A\{\!|q|\!\}@U$ of functions mapping $x : B$ into $\{\!|p|\!\}A\{\!|q|\!\}@U$, where A, p, q may depend on x, is also a complete lattice, with the join operator on functions defined point-wise, and the constant {Unfinished} *function as the unit element.*

The **denotation of judgments** $[\![\Gamma \vdash J]\!]$ (Figure 4) turns FCSL judgments into CiC typing judgments ($\vdash_{\mathsf{CiC}}$). A command specification $\{p\}\ -\ : A\ \{q\}@U$ is denoted by the CiC type $\{\!|p|\!\}A\{\!|q|\!\}@U$, and a procedure specification $\forall x : B.\, \{p\}\ -\ : A\ \{q\}@U$ is denoted by the CiC dependent function type $\forall x : B.\, \{\!|p|\!\}A\{\!|q|\!\}@U$. The Hoare ordering $(p_1, q_1) \sqsubseteq (p_2, q_2)$ judgment is explained in Section D, so here we just literally repeat the definition given there.

The **denotation of commands and procedures** (Figure 5) is subsidiary to that of judgments because the fixed-point construction is indexed by the argument and return types, and the pre- and postconditions. An A-returning command c is denoted by a set of approximating trees in $\mathsf{prog}\ U\ A$, and an A-returning procedure F with argument B is denoted by a set of trees in $B \to \mathsf{prog}\ U\ A$.

In the semantic translation for the sequential composition $[\![x \leftarrow c_1; c_2]\!]$, we employ the fact that the translation $[\![c_2]\!]$ can produce *open* values (*i.e.*, trees t_2), where a variable x of CiC is not bound. We subsequently close them by constructing a continuation k as a CiC function, in an indirect way, that is, quantifying over all possible inputs, hence $\forall x, k\ x \in [\![c_2]\!]$. For the denotational semantics of the hiding statement $[\![\mathsf{hide}_{\Phi,g}\ c]\!]$, we appeal to Lemma 6 (Section F.2), which give a constructive way to construct an elaboration predicate $\widehat{\Phi}$ from a given abstraction function Φ. Since all FCSL program constructors preserve *monotonicity*, the $\mathsf{fix}\ f.x.c$ procedure can take the least fixed point lfp of the function $\lambda f.\, \lambda x.\, [\![c]\!]$ by the Knaster-Tarski theorem.

F.7 Soundness Result

We culminate this section with the proof of soundness of the interpretation. The main soundness theorem (Theorem 3), which we present at the end of this section, is a key result of our formal development. It states that FCSL is sound as a logic with respect to its translation into CiC (*i.e.* shallow embedding), that is, for any judgment J, $\Gamma \vdash J$ implies $[\![\Gamma \vdash J]\!]$. In particular, it means that if a judgment of the form $\Gamma \vdash \{p\}\, c : A\, \{q\}@U$ is derivable in FCSL, then its translation $[\![\Gamma]\!] \vdash_{\mathsf{CiC}} [\![c]\!] : \{\!|p|\!\}A\{\!|q|\!\}@U$ is derivable in CiC. Next, let us apply Definition 23 of a Hoare type to $\{\!|p|\!\}A\{\!|q|\!\}@U$ and recall that $[\![c]\!]$ is just a set of action trees, corresponding to the command c (Figure 5). Therefore, by the derivation under $\vdash_{\mathsf{CiC}}$, all action trees from $[\![c]\!]$ will belong to $\{\!|p|\!\}A\{\!|q|\!\}@U$, which automatically implies that for any state w and each tree from $[\![c]\!]$, if w satisfies p and $w \in \mathcal{W}_U$, then $\mathsf{after}_U\ w\ t\ q$ also holds. The later means (Definition 22) that t is safe and $q\ v\ w'$ eventually holds if t steps completely to Ret v in some final configuration w'. Since after employs the always predicate, it also implies the progress property (Lemma 7) for all action trees from $[\![c]\!]$.

Auxiliary modal lemmas The proof of the soundness result will rely on a number of auxiliary modal lemmas, which are listed below. For lack of space, we omit their proofs. They are relatively straightforward, usually by an induction on the schedule ζ used by always predicate. We have carried all of proofs for these lemmas in Coq [11], they usually proceed by an induction on the schedule ζ. In the statements of the lemmas below, we always assume U to be some fixed concurroid, t ranges over action trees and w ranges over states.

The Universal lemma states that the modal predicate always commutes with universal quantification, which yields to the soundness of an infinitary CONJunction rule (Figure 2). The assumption $\mathsf{always}_U\ w\ t\ (\lambda\ w'\ t'.\ \mathsf{True})$ makes the lemma hold when the quantification over x is vacuous.

Lemma 9 (Universal). *If* $\mathsf{always}_U\ w\ t\ (\lambda\ w'\ t'.\ \mathsf{True})$*, then* always *commutes with universal quantification:*

$$\mathsf{always}_U\ w\ t\ (\lambda w'\ t'.\ \forall x.\ P\ x\ w'\ t') \iff \forall x.\ \mathsf{always}_U\ w\ t\ (\lambda w'\ t'.\ P\ x\ w'\ t')$$

The implication lemma corresponds to weakening the postcondition, which is necessary for the proof of CONSEQ rule.

Lemma 10 (Implication for after). *If* $\mathsf{after}_U\ w\ t\ Q_1$ *and* $Q_1\ v\ w \implies Q_2\ v\ w$*, for all* v *and* $w \in \mathcal{W}_U$*, then* $\mathsf{after}_U\ w\ t\ Q_2$.

Closure under sequential composition justifies the Seq rule: q holds at the end of the composed tree if final configuration of the prefix t_1 can be used as an initial configuration for the suffix to show q holds after.

Lemma 11 (Closure under sequential composition). *For some action tree* $t_1 : \mathsf{tree}\ U\ B$ *and* $K : B \to \mathcal{P}(\mathsf{tree}\ U\ A)$*, if* $t_2 \in \{\mathsf{Seq}\ t_1\ k \mid \forall x.\ k\ x \in K\ x\}$ *and*

$$\mathsf{after}_U\ w_1\ t_1\ (\lambda v\ w.\ \forall t.\ (t \in K\ v) \implies \mathsf{after}_U\ w\ t\ q),$$

then $\mathsf{after}_U\ w_1\ t_2\ q$.

Closure under parallel composition justifies the PAR rule. Intuitively, it holds because when (an approximation t_2 of) c_2 takes a step over its private and shared state, it amounts to $\mathcal{R}_U$ environment interference on (an approximation t_1 of) c_1, and vice versa. Note that the pattern of rearranging subjective *self*/*other* components recurs at the level of triples $w = [s \mid j \mid o]$: the parallel composition uses $[s_1 \circ s_2 \mid j \mid o]$ and the left and right child threads use $[s_1 \mid j \mid o \circ s_2]$ and $[s_2 \mid j \mid o \circ s_1]$, respectively.

Lemma 12 (Closure under parallel composition). *If* $\mathsf{after}_U\ ([s_1 \circ s_2 \mid j \mid o])\ t_1\ Q_1$ *and* $\mathsf{after}_U\ ([s_2 \circ s_1 \mid j \mid o])\ t_2\ Q_2$, *then*

$$\begin{aligned}\mathsf{after}_U\ ([s_1 \circ s_2 \mid j \mid o])\ (\mathsf{Par}\ t_1\ t_2\ (\lambda x.\, \mathsf{Ret}\ x))\\ (\lambda v'\ w'.\, \exists s_1'\ s_2'\ j'\ o'.\ w' = [s_1' \circ s_2' \mid j' \mid o'] \quad \wedge\\ P_1\ \pi_1\ v'\ [s_1' \mid j' \mid o' \circ s_2'] \quad \wedge\\ P_2\ \pi_2\ v'\ [s_2' \mid j' \mid o' \circ s_1']).\end{aligned}$$

Closure under injection is targeted to justify the INJECT rule. It states that if after for some predicate Q holds on a state w_U from a "small" concurroid V, it will hold in a large concurroid $U = V \bowtie W$ with an additional state, reachable by the $\mathcal{R}_W$ relation of an addition W (Definition 17).

Lemma 13 (Closure under injection). *For the states* w_V *and* w_0, *such that* $U = V \bowtie W$, $w_V \in \mathcal{W}_U$, $w_0 \in \mathcal{W}_W$ *and* $w_V \cup w_0 \in \mathcal{W}_V$. *If* $\mathsf{after}_V\ w_V\ t\ Q$, *then*

$$\begin{aligned}\mathsf{after}_U\ (w_V \cup w_0)\ (\mathsf{Inject}\ t)\ (\lambda\ v'\ w'.\ \exists w_V'\ w_0'.\ w' = w_V' \cup w_0' \wedge w_V' \in \mathcal{W}_U \wedge\\ (w_0, w_0') \in \mathcal{R}_W \wedge Q\ v'\ w_V').\end{aligned}$$

Closure under hiding justifies the rule HIDE. It is important to notice that in the final state w_V' of the refined execution in V is related to the final state w_U' of the coarse execution in U by the same refinement function $\widehat{\Phi}$ (Definition 18), but with different auxiliaries g', hence the conjunct $\widehat{\Phi}(g')(w_U', w_V')$.

Lemma 14 (Closure under hiding). *If* $\widehat{\Phi}(g)(w_U, w_V)$, *such that* $\mathsf{refines}\ U\ V\ \widehat{\Phi}$, *and* $\mathsf{after}_V\ w_V\ t\ Q$, *then*

$$\mathsf{after}_U\ w_U\ (\mathsf{Hide}\ \Phi\ g\ t)\ (\lambda v\ w_U'.\ \exists w_V'\ g'.\ \widehat{\Phi}(g')(w_U', w_V') \wedge Q\ v\ w_V')$$

Finally, we can formulate and prove the main theorem. The theorem essentially states that if a judgment $\Gamma \vdash J$ can be derived in FCSL, its translation (Figure 4) can be also derived in CiC.

Theorem 3 (Soundness). *If* $\Gamma \vdash J$, *then* $[\![\Gamma \vdash J]\!]$.

Proof. The proof goes by induction on the derivation of J. The proof for pure expression is straightforward, since $[\![\Gamma \vdash e : A]\!]$ is just $[\![\Gamma]\!] \vdash_{\mathsf{CiC}} e : A$. So let us focus on the judgments of the form $\Gamma \vdash \{p\}\, c : A\, \{q\}@U$ that check the specifications of FCSL commands (Figure 2).

Each basic command (*e.g.*, return, act) is sound because its pre- and postconditions are stable under environment interference, the precondition implies the command

is safe (by Definition 23 and through the after predicate), and the resulting configuration satisfies the postcondition. The SEQ, INJECT and HIDE rules are sound by Lemmas 11, 13 and 14. The PAR rule is sound by Lemma 12, as the *self*/*other* exchange in the lemma accounts exactly for the interpretation of the subjective separate conjunction $\circledast$ (Figure 1), and the rule FRAME is just a particular case of PAR with c_2 being an idle thread (*e.g.*, return ();), hence the stability requirement. The fix rule is sound by the Knaster-Tarski theorem. The CONJ rule is sound by Lemma 9, and the CONSEQ rule by Lemma 10. The EXISTential rule and IF rules are derivable. Since SCSL procedures are interpreted as (monadic) CiC functions. the procedure APPlication and HYPothesis rules are sound by the function application and hypothesis rules of CiC. □