Subjective Auxiliary State for Coarse-Grained Concurrency

Ruy Ley-Wild   Aleksandar Nanevski
IMDEA Software Institute
{ruy.leywild, aleks.nanevski}@imdea.org

Abstract

From Owicki-Gries' Resource Invariants and Jones' Rely/Guarantee to modern variants based on Separation Logic, axiomatic logics for concurrency require auxiliary state to explicitly relate the effect of all threads to the global invariant on the shared resource. Unfortunately, auxiliary state gives the proof of an individual thread access to the auxiliaries of all other threads. This makes proofs sensitive to the global context, which prevents local reasoning and compositionality.

To tame this historical difficulty of auxiliary state, we propose *subjective auxiliary state*, whereby each thread is verified using a *self* view (i.e., the thread's effect on the shared resource) and an *other* view (i.e., the collective effect of all the other threads). Subjectivity generalizes auxiliary state from stacks and heaps to user-chosen partial commutative monoids, which can eliminate the dependence on the global thread structure.

We employ subjectivity to formulate Subjective Concurrent Separation Logic as a combination of subjective auxiliary state and Concurrent Separation Logic. The logic yields simple, compositional proofs of coarse-grained concurrent programs that use auxiliary state, and scales to support higher-order recursive procedures that can themselves fork new threads. We prove the soundness of the logic with a novel denotational semantics of action trees and a definition of safety using rely/guarantee transitions over a large subjective footprint. We have mechanized the denotational semantics, logic, metatheory, and a number of examples by a shallow embedding in Coq.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Program Verification—Correctness proofs; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

Keywords concurrency, verification, local reasoning, rely-guarantee thinking, dependent type theory

1. Introduction

Auxiliary state (*a.k.a.* ghost state) is a necessary evil that compensates for the "expressive weakness" [13] of axiomatic program logics for concurrency. It is necessary in proofs to partially expose the internal behavior of threads in order to relate local program assertions to global invariants, and in the metatheory to establish completeness [20]. But it is evil because the proof of an individual thread becomes sensitive to the global thread structure of the context in which it is used. This prevents local reasoning and limits the compositionality of all logics that use auxiliary state.

$$
\begin{array}{c}
\{y \mathrel{\mapstochar\rightharpoonup} \alpha * z \mathrel{\mapstochar\rightharpoonup} \beta\} \\
\begin{array}{c|c}
\begin{array}{c}
\{y \mathrel{\mapstochar\rightharpoonup} \alpha\} \\
\left\langle \begin{array}{c}
\{y \mathrel{\mapstochar\rightharpoonup} \alpha * \exists\beta. I_2(\alpha,\beta)\} \\
x := [x]+1;\, y := [y]+1; \\
\{y \mathrel{\mapstochar\rightharpoonup} \alpha{+}1 * \exists\beta. I_2(\alpha{+}1,\beta)\}
\end{array} \right\rangle \\
\{y \mathrel{\mapstochar\rightharpoonup} \alpha{+}1\}
\end{array}
&
\begin{array}{c}
\{z \mathrel{\mapstochar\rightharpoonup} \beta\} \\
\left\langle \begin{array}{c}
\{z \mathrel{\mapstochar\rightharpoonup} \beta * \exists\alpha. I_2(\alpha,\beta)\} \\
x := [x]+1;\, z := [z]+1; \\
\{z \mathrel{\mapstochar\rightharpoonup} \beta{+}1 * \exists\alpha. I_2(\alpha,\beta{+}1)\}
\end{array} \right\rangle \\
\{z \mathrel{\mapstochar\rightharpoonup} \beta{+}1\}
\end{array}
\end{array} \\
\{y \mathrel{\mapstochar\rightharpoonup} \alpha{+}1 * z \mathrel{\mapstochar\rightharpoonup} \beta{+}1\}
\end{array}
$$

Figure 1. Two incr threads in Concurrent Separation Logic with fractional permissions.

We illustrate the problem of auxiliary state through the classical concurrent incrementor example of Owicki and Gries. The program $\mathsf{incr}(n) \mathrel{\widehat{=}} \langle x := [x] + n\rangle$ enters a *critical section* $\langle - \rangle$ to gain access to the shared state x and increment it by n. We want to verify incr with a *compositional* specification that can be used to deduce the overall effect on x by several incr's in parallel. For example, if k threads concurrently call $\mathsf{incr}(1)$, we should be able to deduce that the overall effect is for x to increment by k.

In Owicki and Gries' **Resource Invariants** (RI) [21], the shared state is specified by a *resource invariant* I that holds whenever all threads are outside the critical section. By *mutual exclusion*, when a thread enters a critical section it acquires exclusive access to the shared state satisfying the invariant. It may mutate the shared state and violate the invariant, but it must restore the invariant before releasing the resource and leaving the critical section. RI is notably useful for *coarse-grained* concurrency, where interference is confined to critical sections. Because the invariant abstracts away from having to reason explicitly about concurrent interference, RI enables sequential reasoning about individual threads.

An invariant that *only* refers to x, however, has no way to identify each thread's effect on the total value, thus, at best it can show that x increments by an unspecified amount. This suffices to prove memory safety, but not to deduce a noninvariant property such as x increments by k. Owicki and Gries' solution (and most axiomatic logics follow suit) is for each thread to have *auxiliary state* that exposes its effect on the shared state to the invariant; auxiliaries are updated for the sake of verification, but aren't needed for execution.

For two parallel incr threads, in **Concurrent Separation Logic** (CSL) [19] with fractional permissions [1] (Figure 1), the *binary* invariant $I_2(\alpha,\beta) \mathrel{\widehat{=}} x \mapsto \alpha{+}\beta * y \mathrel{\mapstochar\rightharpoonup} \alpha * z \mathrel{\mapstochar\rightharpoonup} \beta$ says that the shared state x contains the sum $\alpha{+}\beta$, auxiliary state y captures the first thread's cumulative effect on (i.e., contribution to) x, and likewise z for the second thread.[1] Ownership of an auxiliary is split in half

POPL'13, January 23–25, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1832-7/13/01. . . $10.00

[1] We take Separation Logic heaps as our model of state. The separating conjunction $p{*}q$ means that the heap splits into disjoint subheaps that satisfy

between the invariant and the owning thread, to ensure that only the owner can update the auxiliary. The program is instrumented with *shaded* code to update auxiliaries, and with {braced} assertions that say what holds at specific program points. The final assertion and the invariant imply that x increments by 2.

However, if instead we wanted to prove the effect of three parallel increments, we would have to use a different *ternary* invariant $I_3(\alpha, \beta, \gamma) \mathrel{\widehat{=}} x \mapsto \alpha{+}\beta{+}\gamma * y \mathrel{\mapstochar\rightharpoonup} \alpha * z \mathrel{\mapstochar\rightharpoonup} \beta * w \mathrel{\mapstochar\rightharpoonup} \gamma$. Even though the two-thread program is a subprogram of the three-thread one, the former's proof is specific to the binary invariant I_2 and cannot be reused in the latter's proof. Thus, in RI and its logical descendants such as CSL, even a simple program as incrementation lacks a single proof that can be reused in different contexts.

In this sense, axiomatic proofs with auxiliary state have limited compositionality: it is not always possible to prove a subprogram and reuse its proof directly in a larger context. The problem is that when an invariant mentions auxiliary state, the auxiliaries reveal thread identities (e.g., w in I_3 identifies the new thread). Therefore, when the verification of a thread sees the invariant in the critical section, it becomes dependent on the global thread structure, which breaks local reasoning. Even worse, when threads are forked dynamically (e.g., an iterator forks a separate thread to process each element in a list), the invariant must be defined in terms of dynamically-allocated auxiliary state that mirrors the forking pattern, which makes the invariant even more context specific.

For *fine-grained* concurrency, where interference may occur between any two memory operations, Jones' **Rely/Guarantee** (RG) [12] explicitly specifies thread and environment interference on the shared state. This can reduce and sometimes even eliminate the auxiliary state required in RI proofs. However, RG and most of its logical descendants (e.g., RGSep [26], SAGL [9], Deny-Guarantee [7]) still require noncompositional auxiliary state for the incrementor. To our knowledge, the only exceptions are Concurrent Abstract Predicates (CAP) [6] and HLRG [10], which *can* verify the incrementor without auxiliary state. However, both logics use RG-style specifications that impose complex reasoning about interference explicitly in the proof. We consider this should be avoided for reasoning about any coarse-grained algorithm, because critical sections are used exactly to abstract away from interference.

We present **Subjective Concurrent Separation Logic** (SCSL), a prototype logic for a stateful language with higher-order recursive procedures, fork-join concurrency, and a lock-protected critical section (Section 4). SCSL combines the classical techniques of Resource Invariants and Separation Logic, and tames the difficulties of auxiliary state with a novel *subjective* perspective.

Subjective auxiliary state gives two complementary thread-oriented views on auxiliary state: a *self* view $\mathsf{a_S}$ of the thread's own auxiliary contribution, and an *other* view $\mathsf{a_O}$ that agglomerates all other threads' contributions. To eliminate the dependency on the number of threads in the environment, subjective auxiliary state generalizes from stack variables and heap locations to a user-chosen *partial commutative monoid* (PCM). For various examples, choosing an optimized PCM can blur the global thread structure, in contrast to auxiliary stacks or heaps. Since stacks and heaps are PCMs [4], subjective auxiliary state subsumes classical auxiliary state. Thus, when a suitable auxiliary PCM cannot be found, a proof can always fall back on using an auxiliary stack or heap.

Intuitively, subjectivity synthesizes the thread/environment dichotomy espoused by Rely/Guarantee and the PCM structure of state elucidated by Separation Logic. In contrast to the *temporal* emphasis of Rely/Guarantee transitions, subjective auxiliary state creates a *spatial* dichotomy between thread and environment auxiliaries. In contrast to the exclusively *local*, small footprint, view of state in Separation Logic, the $\mathsf{a_O}$ view of the environment embodies *non-local* information about the run-time context, but the PCM-agglomeration makes it abstract enough to retain the CSL-style local reasoning.

The critical point concerns how to combine proofs of two threads into a proof of their **parallel composition**. When one thread updates its $\mathsf{a_S}$, then it implicitly also updates another thread's $\mathsf{a_O}$. How do we ensure that the proof of the other thread remains a valid subproof in the combination, when it sees its $\mathsf{a_O}$ view change?

We resolve this question by proving a CSL-style rule of parallel composition with a novel *subjective separating conjunction* connective $\circledast$ for splitting auxiliary views. The rule ensures that if the initial views of two threads are coherent at the fork point, then there exists a coherent combination of the views at the join point. This suffices to use the proofs of threads as black boxes and establish the combined verification of their parallel composition. We also prove a frame rule with $\circledast$ that surprisingly does not require cancellativity of the auxiliary PCM.

We show that SCSL yields simple, compositional proofs for a few coarse-grained concurrent programs from the literature (Sections 2 and 3). Although the programs are relatively simple, their proofs in existing logics either require auxiliaries which leads to noncompositionality, or can avoid auxiliaries but are technically complex. Moreover, we show that SCSL scales to higher-order recursive procedures which can themselves fork new thread, depending on run time values; to our knowledge, this combination is not supported by any existing logic[2]. Although we focus on RI proofs for coarse-grained concurrency, we believe subjectivity will also synergize well with RG for fine-grained concurrency.

We give SCSL a novel denotational **model** that represents computations as *action trees* and establish **soundness** of the logic by a definition of safety that crucially relies on subjectivity, not only of the user-chosen auxiliary PCM, but also of heap and lock ownership (Section 5). We embed SCSL into the higher-order logic Calculus of Inductive Constructions [18], and thus inherit a number of features from dependent type theory: abstraction over programs (i.e., higher-order procedures), types (i.e., polymorphism), and predicates (i.e., datatypes, objects, modules). These features are essential for modular programming and verification, but, aside from abstract predicates [22, 6], have not been considered in the axiomatic concurrency logic literature. The embedding allowed us to mechanize the semantics, logic, soundness proof, and a number of examples in Coq; the Coq sources are available online [16].

2. SCSL and Proving the Incrementor

2.1 Subjective Auxiliary State

Subjective Concurrent Separation Logic (SCSL) has a single lock with global scope which is associated with a user-chosen partial commutative monoid (PCM) $(\mathbb{U}, \oplus, \mathbb{0})$ for auxiliary state and an invariant I_* indexed by $\mathbb{U}$. A PCM is a set $\mathbb{U}$ with an associative and commutative join operation $\oplus$ and unit element $\mathbb{0}$. Intuitively, the unit $\mathbb{0}$ describes a null effect; the join $\oplus$ combines the effects of two parallel threads, which mirrors the associativity and commutativity of parallel composition; and partiality indicates inconsistent views (e.g., two threads owning a lock simultaneously).

p and q that are passed to each subthread of a parallel composition; $\ell \mathrel{\mapstochar\rightharpoonup} v$ is a *half fractional permission* that allows dereferencing ℓ with contents v; $\ell \mathrel{\mapstochar\rightharpoonup} v * \ell \mathrel{\mapstochar\rightharpoonup} v$ is equivalent to $\ell \mapsto v$, which also allows updating ℓ. Strictly speaking, the invariant is the unparameterized assertion $\exists\alpha.\exists\beta.I_2(\alpha, \beta)$ where the auxiliary contributions α and β are existentially quantified; we concentrate on the parameterized I_2 to simplify the presentation and omit the details of how fractional permissions relate the free and bound variables.

[2] We don't, however, support higher-order heaps that can store procedures.

By working with an auxiliary PCM, we can take the $\oplus$-total $\tau \in \mathbb{U}$ of *all* threads' combined contributions: if each thread i contributes σ_i, then $\tau = \bigoplus_i \sigma_i$. The single total auxiliary value τ serves to define a *unary* invariant $I_*(\tau)$ that doesn't depend on the number of threads, in contrast to CSL proofs where auxiliaries expose all threads in the invariant (e.g., $y \mapsto \alpha$ identifies the left thread and $z \mapsto \beta$ the right thread in the CSL proof from Section 1).

To preserve locality, however, assertions can't directly refer to the total τ, because all threads can interfere on its value. Instead, each thread i has two complementary subjective *views* of τ: a *self* view $\mathsf{a_S} \rightarrowtail \sigma_i$ of the thread's own auxiliary contribution, and an *other* view $\mathsf{a_O} \hookrightarrow \omega_i$ where $\omega_i = \bigoplus_{j \neq i} \sigma_j$ is the $\oplus$-subtotal of all other threads' contributions.[3] Although different threads i and j may have different views $(\mathsf{a_S}, \mathsf{a_O})$, all views are mutually *coherent* in that they yield the same $\oplus$-total: $\sigma_i \oplus \omega_i = \tau = \sigma_j \oplus \omega_j$. We use $\mathsf{a_S} \rightarrowtail \sigma_i$ throughout the program to track the thread's contribution as it changes, but only use $\mathsf{a_O} \hookrightarrow \omega_i$ in the critical section to track that the environment's contribution remains fixed when it can't access the shared resource (Section 2.4).

2.2 Parallel Composition and Subjective Separation

For the incrementor, we choose the PCM $(\mathbb{N}, +, 0)$ of natural numbers under addition and the unary invariant $I_*(\tau) \mathrel{\widehat{=}} x \mapsto \tau$, which hides the identity of threads and thus permits a compositional proof of incr. The SCSL specification of $\mathsf{incr}(n)$ is:

$$\forall n{:}\mathsf{nat}.\{\mathsf{a_S} \rightarrowtail \sigma\}\mathsf{incr}(n) : \mathsf{unit}\{\mathsf{a_S} \rightarrowtail \sigma + n\}$$

incr takes an argument n and increments the thread's contribution $\mathsf{a_S}$ from σ to $\sigma + n$,[4] and returns unit.[5] Since the value of $\mathsf{a_S}$ is thread-specific, the SCSL specification does not expose thread identities (*cf.* how y identified the left thread in the CSL proof).

The environment's contribution $\mathsf{a_O}$ does not appear in the specification and is thus unconstrained, but it will show up in the proof of incr. Since incr only mutates shared state, the local heap is implicitly empty in the pre and post. By framing, incr is safe to run in any larger private heap, which remains unchanged upon termination.

To prove two parallel increments, we use the SCSL PAR rule:

$$\frac{\{p_1\}C_1\{q_1\} \qquad \{p_2\}C_2\{q_2\}}{\{p_1 \circledast p_2\}C_1 \parallel C_2\{q_1 \circledast q_2\}}\ \textsc{Par}$$

The ***subjective separating conjunction*** $\circledast$ connective splits the heap into two disjoint subheaps; the novelty is that $\circledast$ also splits the *contents* of $\mathsf{a_S}$: $\mathsf{a_S} \rightarrowtail \alpha \circledast \mathsf{a_S} \rightarrowtail \beta$ is equivalent to $\mathsf{a_S} \rightarrowtail \alpha \oplus \beta$. For comparison, the CSL PAR rule uses the separating conjunction $*$ to split the heap and *ownership* of auxiliaries among parallel threads.

The SCSL proof of incr is compositional: we can instantiate σ and n in the proof of incr, which is otherwise used as a black box, to show that $\mathsf{incr}(i) \parallel \mathsf{incr}(j)$ increments x by $i+j$.

$$\begin{array}{c}
\{\mathsf{a_S} \rightarrowtail \alpha{+}\beta\} \\
\{\mathsf{a_S} \rightarrowtail \alpha \circledast \mathsf{a_S} \rightarrowtail \beta\} \\
\begin{array}{c|c}
\{\mathsf{a_S} \rightarrowtail \alpha\} & \{\mathsf{a_S} \rightarrowtail \beta\} \\
\mathsf{incr}(i) & \mathsf{incr}(j) \\
\{\mathsf{a_S} \rightarrowtail \alpha{+}i\} & \{\mathsf{a_S} \rightarrowtail \beta{+}j\}
\end{array} \\
\{\mathsf{a_S} \rightarrowtail \alpha{+}i \circledast \mathsf{a_S} \rightarrowtail \beta{+}j\} \\
\{\mathsf{a_S} \rightarrowtail (\alpha{+}i){+}(\beta{+}j)\}
\end{array}$$

The parallel composition starts with a contribution $\mathsf{a_S} \rightarrowtail \alpha{+}\beta$. When the parent thread forks two child threads, the parent's view $\mathsf{a_S}$ splits between its children, while the children's $\mathsf{a_O}$'s are implicitly induced to preserve coherence (i.e., the left child's environment includes the right child, and vice versa). By coherence, the combination of views is the same irrespective of the viewer.[6]

viewer:		parent $\Leftrightarrow$	child$_1$ $\circledast$	child$_2$
views:	self $\mathsf{a_S}$ (explicit) other $\mathsf{a_O}$ (implicit)	$\alpha{+}\beta$ ω	α $\beta{+}\omega$	β $\alpha{+}\omega$
combined views:	$\mathsf{a_S}{+}\mathsf{a_O}$	$\alpha{+}\beta{+}\omega$		

The left thread instantiates the incr proof with α and i to show that the overall contribution increments from α to $\alpha{+}i$; and analogously for the right thread. The soundness of parallel composition ensures that, dually, at the join point the children's $\mathsf{a_S}$'s can be combined back into the parent's $\mathsf{a_S}$: the subthreads' $\mathsf{a_S}$'s are added and the parallel composition ends with $\mathsf{a_S}$ incremented by $i{+}j$ over the initial value. $\mathsf{a_O}$ is implicitly unconstrained and unknown in the parent, and thus remains so in both child threads.

We can directly reuse the two-thread SCSL proof compositionally in a three-thread program *without having to change the invariant or subproof for the particular context*. The three-thread program increments the initial contribution $\alpha{+}\beta{+}\gamma$ by $i{+}j{+}k$ with a structurally similar proof as the two-thread program.

$$\begin{array}{c}
\{\mathsf{a_S} \rightarrowtail \alpha{+}\beta{+}\gamma\} \\
\{\mathsf{a_S} \rightarrowtail \alpha{+}\beta \circledast \mathsf{a_S} \rightarrowtail \gamma\} \\
\begin{array}{c|c}
\{\mathsf{a_S} \rightarrowtail \alpha{+}\beta\} & \{\mathsf{a_S} \rightarrowtail \gamma\} \\
\mathsf{incr}(i) \parallel \mathsf{incr}(j) & \mathsf{incr}(k) \\
\{\mathsf{a_S} \rightarrowtail (\alpha{+}i){+}(\beta{+}j)\} & \{\mathsf{a_S} \rightarrowtail \gamma{+}k\}
\end{array} \\
\{\mathsf{a_S} \rightarrowtail (\alpha{+}i){+}(\beta{+}j) \circledast \mathsf{a_S} \rightarrowtail \gamma{+}k\} \\
\{\mathsf{a_S} \rightarrowtail (\alpha{+}i){+}(\beta{+}j){+}(\gamma{+}k)\}
\end{array}$$

Here, the initial $(\mathsf{a_S}, \mathsf{a_O})$ views are $(\alpha{+}\beta{+}\gamma, \omega)$ for some ω, which induce views for the three threads: $(\alpha, \beta{+}\gamma{+}\omega)$, $(\beta, \alpha{+}\gamma{+}\omega)$, and $(\gamma, \alpha{+}\beta{+}\omega)$; all have the coherent total $\alpha{+}\beta{+}\gamma{+}\omega$.

Furthermore, we can substitute programs with the same specification. For example, the parallel $\mathsf{incr}(i) \parallel \mathsf{incr}(j)$ and the sequential $\mathsf{incr}(i); \mathsf{incr}(j)$ with a different number of threads can be given the same specification $\{\mathsf{a_S} \rightarrowtail \alpha{+}\beta\} - \{\mathsf{a_S} \rightarrowtail \alpha{+}\beta{+}i{+}j\}$. Therefore we can use them interchangeably in the three-increment program and the overall proof remains the same. In Section 3.2, we show that this approach scales to prove a concurrent iterator, where the number of threads depends on a run-time value.

2.3 Assertions

In SCSL, an assertion is a predicate over a *world* w of the form $h; \mu; \sigma; \omega$. h is the thread's private heap (a finite map from pointers to values), which is a PCM with the empty heap as unit and disjoint

$h;\mu;\sigma;\omega \models \mathsf{a_S} \rightarrowtail \sigma'$	iff $(h,\mu,\sigma) = (\mathsf{empty}, \cancel{\mathsf{Own}}, \sigma')$
$h;\mu;\sigma;\omega \models \mathsf{m_S} \rightarrowtail \mu'$	iff $(h,\mu,\sigma) = (\mathsf{empty}, \mu', \mathbb{0})$
$h;\mu;\sigma;\omega \models \ell \mapsto v$	iff $(h,\mu,\sigma) = ([\ell \Mapsto v], \cancel{\mathsf{Own}}, \mathbb{0})$
$h;\mu;\sigma;\omega \models \mathsf{a_O} \hookrightarrow \omega'$	iff $\omega = \omega'$
$h;\mu;\sigma;\omega \models \mathsf{emp}$	iff $(h,\mu,\sigma) = (\mathsf{empty}, \cancel{\mathsf{Own}}, \mathbb{0})$
$h;\mu;\sigma;\omega \models p_1 * p_2$	iff $h_1;\mu_1;\sigma_1;\omega \models p_1$ and $h_2;\mu_2;\sigma_2;\omega \models p_2$ where $(h,\mu,\sigma) = (h_1 \oplus h_2, \mu_1 \oplus \mu_2, \sigma_1 \oplus \sigma_2)$ for some h_i, μ_i, σ_i
$h;\mu;\sigma;\omega \models p_1 \circledast p_2$	iff $h_1;\mu_1;\sigma_1;\sigma_2 \oplus \omega \models p_1$ and $h_2;\mu_2;\sigma_2;\sigma_1 \oplus \omega \models p_2$ where $(h,\mu,\sigma) = (h_1 \oplus h_2, \mu_1 \oplus \mu_2, \sigma_1 \oplus \sigma_2)$ for some h_i, μ_i, σ_i
$w \models p_1 \wedge p_2$	iff $w \models p_1$ and $w \models p_2$
$[\mu, \sigma, \omega]\, p$ abbreviates	$(\mathsf{m_S} \rightarrowtail \mu * \mathsf{a_S} \rightarrowtail \sigma * p) \wedge \mathsf{a_O} \hookrightarrow \omega$

Figure 2. Semantics of assertions $h; \mu; \sigma; \omega \models p$ (fragment).

[3] For the moment, $\mathsf{a_S} \rightarrowtail \sigma$ and $\mathsf{a_O} \hookrightarrow \omega$ should be read as asserting the contents of each auxiliary view, but don't directly permit mutation; their formal meaning is explained in Section 2.3.

[4] We use Greek letters, such as σ, for *logical variables* that scope over the pre- and postcondition to relate initial and final states.

[5] The SCSL judgments include an explicit return type (Section 4). Hereafter we omit the return type unless it requires commentary.

[6] This pattern recurs in the semantics, not only for auxiliaries but also for heaps and lock ownership (Section 5.5).

union $\uplus$ as join (undefined if the heaps overlap). μ is an element of the lock ownership PCM mtx, with elements Own (to indicate the viewer owns the lock and is thus in a critical section) and $\not{\mathsf{Own}}$; the join $\oplus$ is defined with unit $\not{\mathsf{Own}}$, but $\mathsf{Own} \oplus \mathsf{Own}$ is undefined because two threads cannot enter a critical section simultaneously. σ and ω are the thread and environment's respective auxiliary contributions from $\mathbb{U}$.

In our subjective terminology, the components h, μ and σ are the *self* views on the thread's world, as they describe the state of the thread itself. On the other hand, ω is the *other* view, as it describes the abstract effect of the thread's environment on the shared state.

Figure 2 gives the interpretation of assertions.[7] The thread's auxiliary pointer assertion $\mathsf{a_S} \rightarrowtail \sigma'$ constrains the σ component of the world to be σ', and the h and μ components to be the respective PCM units, but the ω component is unconstrained because the environment may enter the critical section and change its contribution in tandem with the shared state. Thus, the $\mathsf{a_S} \rightarrowtail -$ assertions in the specification of incr implicitly mean the private heap is empty and the lock isn't owned. The pointer assertion $\ell \mapsto v$ and lock (*a.k.a.* mutex) pointer assertion $\mathsf{m_S} \rightarrowtail \mu'$ are defined analogously. The environment's auxiliary pointer assertion $\mathsf{a_O} \hookrightarrow \omega'$ constrains the ω component of the world to be ω', but is *intuitionistic* in that it doesn't constrain the h, μ, or σ components. The auxiliary pointer assertions use a half arrow tip to suggest that they can't be mutated directly (though we don't use permissions).

The $p_1 \circledast p_2$ assertion splits the self components h, μ, and σ between p_1 and p_2, and induces the other component to maintain coherence (i.e., $\sigma_2 \oplus \omega$ for p_1, and dually for p_2). The $p_1 * p_2$ assertion also splits the self components h, μ, and σ, but doesn't change the other component ω.

The SCSL rules for memory and locking commands typically use the assertion $[\mu', \sigma', \omega']\, p$, which abbreviates $(\mathsf{m_S} \rightarrowtail \mu' * \mathsf{a_S} \rightarrowtail \sigma' * p) \wedge \mathsf{a_O} \hookrightarrow \omega'$ and holds in a world $h; \mu; \sigma; \omega$ if $(\mu, \sigma, \omega) = (\mu', \sigma', \omega')$ and h satisfies the heap predicate p. The asymmetry between the $\mathsf{a_S} \rightarrowtail -$ and $\mathsf{a_O} \hookrightarrow -$ assertions stems from the fact that $\mathsf{a_S}$ must be tracked throughout the program, in particular when it is split at forking points, but $\mathsf{a_O}$ is implicitly induced to preserve coherence at forking points and is only tracked in the critical section.

2.4 Critical Sections and Local Auxiliary Functions

The following proof outline of incr is parametric in σ_i and n. In SCSL, we enter and exit *critical sections* using explicit lock and unlock commands, which illustrates how lock ownership works in tandem with auxiliaries.

```
   $\{\mathsf{a_S} \rightarrowtail \sigma_i\}$                                            precondition
   $\{[\not{\mathsf{Own}}, \sigma_i, -]\, \mathsf{emp}\}$                                 by ConSeq
1  $\mathsf{lock};$
   $\{\exists \omega_i . [\mathsf{Own}, \sigma_i, \omega_i]\, I_*(\sigma_i + \omega_i)\}$                       by lock
2  $tmp \leftarrow \mathsf{read}\ x;$
   $\{\exists \omega_i . [\mathsf{Own}, \sigma_i, \omega_i]\, x \mapsto (\sigma_i + \omega_i) \wedge tmp = \sigma_i + \omega_i\}$ by read
3  $\mathsf{write}\ x\ (tmp + n);$
   $\{\exists \omega_i . [\mathsf{Own}, \sigma_i, \omega_i]\, x \mapsto ((\sigma_i + \omega_i) + n)\}$               by write
   $\{\exists \omega_i . [\mathsf{Own}, \sigma_i, \omega_i]\, I_*((\sigma_i + n) + \omega_i)\}$                  by ConSeq
4  $\mathsf{unlock}_{\Phi_n}$
   $\{\exists \omega_i . [\not{\mathsf{Own}}, \Phi_n(\sigma_i), -]\, \mathsf{emp}\}$                        by unlock
   $\{\mathsf{a_S} \rightarrowtail \sigma_i + n\}$                                        by ConSeq
```

First, the precondition $\mathsf{a_S} \rightarrowtail \sigma_i$ is turned into the equivalent explicit assertion $[\not{\mathsf{Own}}, \sigma_i, -]\, \mathsf{emp}$, where "$-$" indicates that the environment's contribution is existentially quantified and thus unknown.

The lock operation (line 1) uses the SCSL rule:

$$\{[\not{\mathsf{Own}}, \sigma_i, -]\, \mathsf{emp}\}\mathsf{lock}\{\exists \omega_i . [\mathsf{Own}, \sigma_i, \omega_i]\, I_*\ (\sigma_i \oplus \omega_i)\}$$

where the PCM operation $\oplus$ is instantiated with $+$. Locking changes the ownership status from $\not{\mathsf{Own}}$ to Own and preserves the thread's contribution σ_i. Prior to locking, the environment may enter the critical section and change its contribution in tandem with the shared state, thus $\mathsf{a_O}$ is unconstrained in the precondition and existentially named ω_i in the postcondition. The thread also acquires the shared state satisfying $I_*(\tau)$, which is equivalent to $I_*(\sigma_i \oplus \omega_i)$ by the coherence of views.

By mutual exclusion, other threads j can't access or change the shared resource, so their contributions σ_j and consequently $\mathsf{a_O} \hookrightarrow \omega_i (= \bigoplus_{j \neq i} \sigma_j)$ remain fixed throughout the critical section. This invariance is explicitly specified by the commands in the critical section. For example, the SCSL read rule:

$$\{[\mu_i, \sigma_i, \omega_i]\, x \mapsto \nu\}\mathsf{read}\ x \left\{ \begin{array}{c} [\mu_i, \sigma_i, \mathsf{if}\ \mu = \mathsf{Own}\ \mathsf{then}\ \omega_i\ \mathsf{else}\ -] \\ x \mapsto \nu \wedge \mathsf{res} = \nu \end{array} \right\}$$

includes the standard Separation Logic specification that initial heap contains a pointer x with value ν; in the postcondition, the heap is preserved and ν is bound to the dedicated result variable. Moreover, the lock status μ_i and thread contribution σ_i may be arbitrary and are preserved across the read. However, the environment's contribution is subject to a conditional on the lock ownership: if the thread Owns the lock because it's in a critical section, then $\mathsf{a_O}$ remains fixed at ω_i, otherwise $\mathsf{a_O}$ is undetermined in the postcondition. The other memory commands determine the heap as in Separation Logic and preserve the auxiliaries just like read.

By sequential composition, the read (line 2) binds the contents of x (i.e., $\sigma_i + \omega_i$ by the invariant) to the local variable tmp, and extends the assertion with an equation that describes tmp. The write (line 3) increments x by n, which is reflected in the assertion about the contents of x. Since the thread Owns the lock, the read and write preserve the auxiliaries as σ_i and ω_i.

That the environment's auxiliary $\mathsf{a_O} \hookrightarrow \omega_i$ is fixed throughout the critical section becomes crucial at the unlock command when the invariant $I_*(\sigma'_i \oplus \omega_i)$ must be restored for some *new* thread contribution σ'_i and the *same* environment contribution ω_i from the moment of locking. This emulates the Owicki-Gries discipline that an auxiliary is only updated by its owner in the critical section.

Since we generalize auxiliary state from heap to PCM $\mathbb{U}$, we must also generalize from stack- and heap-mutating auxiliary code to *auxiliary functions* Φ on $\mathbb{U}$. The unlock rule:

$$\{[\mathsf{Own}, \sigma_i, \omega_i]\, I_*\ (\Phi(\sigma_i) \oplus \omega_i)\}\mathsf{unlock}_{\Phi}\{[\not{\mathsf{Own}}, \Phi(\sigma_i), -]\, \mathsf{emp}\}$$

requires the unlock command to be decorated by a mathematical function Φ such that the heap initially satisfies the invariant $I_*(\Phi(\sigma_i) \oplus \omega_i)$ relative to a *new* thread contribution $\Phi(\sigma_i)$, rather than the former thread contribution σ_i. In the postcondition, lock ownership and the invariant heap are released, the thread's auxiliary is updated from σ_i to $\sigma'_i \mathrel{\widehat{=}} \Phi(\sigma_i)$ (which implicitly changes other threads' ω_j). The environment's auxiliary again becomes undetermined, because the thread is no longer in the critical section and the environment can change its contribution at will.

In the incrementor proof, the auxiliary function on unlock (line 4) is $\Phi_n(\sigma) \mathrel{\widehat{=}} \sigma{+}n$, which corresponds to the auxiliary update code used in Section 1 to increment the auxiliaries y and z. By unlocking, the thread's auxiliary $\mathsf{a_S}$ is updated from σ_i to $\Phi_n(\sigma_i) = \sigma_i{+}n$ and the invariant subheap is released. Finally, the existential ω_i can be eliminated because it doesn't appear in the assertion, and we return to the more succinct $\mathsf{a_S} \rightarrowtail \sigma_i + n$.

The structured form of auxiliary update via unlock_{Φ} has several advantages when compared to the classical auxiliary code from Section 1. First, since Φ is a pure function, it can't perform side

[7] We write $[\ell \Mapsto v]$ for the singleton heap that maps ℓ to v, $[h|\ell \Mapsto v]$ for the heap that maps ℓ to v and behaves like h on other locations, and $h \setminus \ell$ for the heap h without ℓ. By taking the lock and auxiliary units $h; \not{\mathsf{Own}}; \mathbb{0}; \mathbb{0} \models p$ we recover the standard Separation Logic relation $h \models_{\mathsf{SL}} p$.

effects to leak the contents of auxiliary state into the heap and thus change the program's semantics.

Second, Φ doesn't directly name $\mathsf{a_S}$. Instead, the proof implicitly applies Φ to the $\mathsf{a_S}$ value of the viewing thread. Thus, Φ can't possibly mutate other threads' auxiliary contributions. In this sense, SCSL differs from a number of related systems [2, 23, 24] which require ownership-tracking permissions to control which auxiliaries can be changed by what auxiliary code (e.g., how the left thread updates y and the right thread updates z in Section 1).

Most importantly, Φ can update the appropriate $\mathsf{a_S}$ without knowing the viewing thread's identity, whereas the CSL proof updates specific auxiliaries in each thread (e.g., y in the left thread, and z in the right thread in Section 1). The independence from the thread's identity facilitates reusability: the *same invariant, auxiliary code, and proof of* incr *can be used without any change*, in contexts with two or three concurrent threads (Section 2.2), or with any number of threads as we illustrate with the iterator (Section 3.2).

To ensure the soundness of parallel composition and framing, however, the auxiliary function Φ must be *local* in the following formal sense, where $\mathsf{valid}(\alpha)$ means α is a defined PCM element. Obviously, the incrementor's Φ_n is local.

Definition 1. *A partial function Φ on a PCM $\mathbb{U}$ is* local, *written $\Phi : \mathbb{U} \rightharpoonup_{\mathsf{L}} \mathbb{U}$, if for all $\alpha, \beta \in \mathbb{U}$,* $\mathsf{valid}\,(\alpha \oplus \beta)$ *and* $\mathsf{valid}\,(\Phi(\alpha))$ *imply $\Phi\,(\alpha \oplus \beta) = \Phi(\alpha) \oplus \beta$.*

Intuitively, Φ can be applied to $\mathsf{a_S}$ with $\mathsf{a_O}$ "framed" on by $\oplus$-combination: it will act on $\mathsf{a_S}$ but simply propagate $\mathsf{a_O}$ without change. This property makes Φ appropriate for modeling auxiliary code for a thread to modify its auxiliary $\mathsf{a_S}$, but is unaffected by and doesn't affect the auxiliary $\mathsf{a_O}$ of other threads running in parallel.

3. More Examples

3.1 Coarse-Grained Set with Disjoint Interference

We adapt a coarse-grained set example from Concurrent Abstract Predicates [6] (in Section 6 we compare with this approach). If threads perform *logically* disjoint operations (i.e., add or remove distinct elements), then we can prove their parallel composition with local reasoning. This is possible even if the operations interfere *physically* on the data structure; for example, if the set is implemented with a splay tree, the memory shape will depend on the order of operations. Fortunately, at the level of abstraction of set membership, the proof isn't sensitive to the noncommutativity of physical operations. Alternatively, we could specify the logical disjointness using classical auxiliary state, but it leads to the same problems as in the incrementor.

We assume a set library, whose procedures sadd and sremove are sequential and can be run in a critical section preserving the auxiliaries:

$$\begin{array}{cc}
\{v \notin \Sigma \wedge [\mathsf{Own}, \sigma, \omega]\ \mathsf{set}(x, \Sigma)\} & \{[\mathsf{Own}, \sigma, \omega]\ \mathsf{set}(x, \Sigma \uplus \{v\})\} \\
\mathsf{sadd}(x, v) & \mathsf{sremove}(x, v) \\
\{[\mathsf{Own}, \sigma, \omega]\ \mathsf{set}(x, \Sigma \uplus \{v\})\} & \{[\mathsf{Own}, \sigma, \omega]\ \mathsf{set}(x, \Sigma)\}
\end{array}$$

Here v is drawn from some universe V of values, and $\mathsf{set}(x, \Sigma)$ is a spatial predicate that means the finite subset $\Sigma \subseteq V$ is represented at pointer x. We can obtain a coarse-grained concurrent library by wrapping each procedure in a critical section:

$$\begin{array}{l}
\mathsf{cadd}(v) \mathrel{\widehat{=}} \mathsf{lock};\ \mathsf{sadd}(x, v) \qquad ;\ \mathsf{unlock} \\
\mathsf{cremove}(v) \mathrel{\widehat{=}} \mathsf{lock};\ \mathsf{sremove}(x, v);\ \mathsf{unlock}
\end{array}$$

Although the coarse-grained implementation is inefficient because threads must contend for the entire data structure, we can choose suitable auxiliaries $\mathsf{in}(v)$ and $\mathsf{out}(v)$ to specify the procedures in a *local* manner that only depends on the element being manipulated:

$$\begin{array}{l}
\{\mathsf{a_S} \rightarrowtail \mathsf{out}(v)\}\ \ \mathsf{cadd}(x, v)\ \ \{\mathsf{a_S} \rightarrowtail \mathsf{in}(v)\} \\
\{\mathsf{a_S} \rightarrowtail \mathsf{in}(v)\}\ \mathsf{cremove}(x, v)\ \{\mathsf{a_S} \rightarrowtail \mathsf{out}(v)\}
\end{array}$$

We can thus reason locally in each thread to prove the effect of concurrently adding or removing three distinct elements:

$$\begin{array}{c}
\{\mathsf{a_S} \rightarrowtail (\mathsf{in}(a) \oplus \mathsf{out}(b) \oplus \mathsf{in}(c))\} \\
\begin{array}{c|c|c}
\{\mathsf{a_S} \rightarrowtail \mathsf{in}(a)\} & \{\mathsf{a_S} \rightarrowtail \mathsf{out}(b)\} & \{\mathsf{a_S} \rightarrowtail \mathsf{in}(c)\} \\
\mathsf{cremove}(a) & \mathsf{cadd}(b) & \mathsf{cremove}(c) \\
\{\mathsf{a_S} \rightarrowtail \mathsf{out}(a)\} & \{\mathsf{a_S} \rightarrowtail \mathsf{in}(b)\} & \{\mathsf{a_S} \rightarrowtail \mathsf{out}(c)\}
\end{array} \\
\{\mathsf{a_S} \rightarrowtail (\mathsf{out}(a) \oplus \mathsf{in}(b) \oplus \mathsf{out}(c))\}
\end{array}$$

To verify the concurrent library in SCSL, the auxiliaries should maintain the distributed, partitioned knowledge of which elements are in or out of the data structure. We pick the auxiliary PCM $(V \rightharpoonup_{\mathrm{fin}} \mathsf{bool}, \uplus, \emptyset)$ of finite partial maps from V to bool, with the join $\uplus$ taking the union of maps with disjoint domain, and the empty map $\emptyset$ as unit. Given such a finite map φ, the invariant $I_*\ (\varphi) \mathrel{\widehat{=}} \mathsf{set}(x, \varphi^{-1}[\mathsf{true}])$ states that the set contains exactly the elements that φ maps to true. We define $\mathsf{in}(v), \mathsf{out}(v) : V \rightharpoonup_{\mathrm{fin}} \mathsf{bool}$ to map v to true and false, respectively, and be undefined elsewhere.

Then we can prove cadd (and cremove similarly) as follows:

$$\begin{array}{ll}
\{\mathsf{a_S} \rightarrowtail \mathsf{out}(v)\} & \text{precondition} \\
\{[\cancel{\mathsf{Own}}, \mathsf{out}(v), -]\ \mathsf{emp}\} & \text{by C}\textsc{onseq} \\
\mathsf{lock}; & \\
\{\exists \varphi_{\mathsf{O}}.[\cancel{\mathsf{Own}}, \mathsf{out}(v), \varphi_{\mathsf{O}}]\ I_*(\mathsf{out}(v) \uplus \varphi_{\mathsf{O}})\} & \text{by L}\textsc{ock} \\
\mathsf{sadd}(x, v); & \\
\{\exists \varphi_{\mathsf{O}}.[\cancel{\mathsf{Own}}, \mathsf{out}(v), \varphi_{\mathsf{O}}]\ I_*(\mathsf{in}(v) \uplus \varphi_{\mathsf{O}})\} & \text{by } \mathsf{sadd} \\
\mathsf{unlock}_{\Phi_v}; & \\
\{[\cancel{\mathsf{Own}}, \mathsf{in}(v), -]\ \mathsf{emp}\} & \text{by U}\textsc{nlock} \\
\{\mathsf{a_S} \rightarrowtail \mathsf{in}(v)\} & \text{by C}\textsc{onseq}
\end{array}$$

When the program locks, the environment's auxiliary φ_{O} comes into scope. The initial invariant $I_*(\mathsf{out}(v) \uplus \varphi_{\mathsf{O}})$ is equivalent to sadd's precondition with $\Sigma{=}(\mathsf{out}(v) \uplus \varphi_{\mathsf{O}})^{-1}[\mathsf{true}]{=}\varphi_{\mathsf{O}}^{-1}[\mathsf{true}]$; sadd's postcondition $\mathsf{set}(x, \varphi_{\mathsf{O}}^{-1}[\mathsf{true}] \uplus \{v\})$ restores the invariant $I_*(\mathsf{in}(v) \uplus \varphi_{\mathsf{O}})$. Unlocking turns $\mathsf{a_S}$ into $\Phi_v(\mathsf{out}(v)){=}\mathsf{in}(v)$, where

$$\Phi_v(\varphi) \mathrel{\widehat{=}} \lambda u. \begin{cases} \mathsf{true} & u = v \text{ and } v \in \mathsf{dom}\ \varphi \\ \varphi(u) & u \neq v \text{ and } u, v \in \mathsf{dom}\ \varphi \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

Φ_v turns a finite map φ with $v \in \mathsf{dom}\ \varphi$ into a φ' that behaves like φ on all elements, except that it maps v to true; if $v \notin \mathsf{dom}\ \varphi$, then $\Phi_v(\varphi)$ is undefined. It is straightforward to check that Φ_v is local.

3.2 Higher-order Iterator

We can prove a *higher-order* iterator that *dynamically forks* a procedure f (itself possibly concurrent) to process each element in a list.[8] In SCSL, we can prove the iterator with a *hypothetical* f:

$$\begin{array}{l}
\forall n{:}A.\ \{\mathsf{a_S} \rightarrowtail \psi(n)\} f(n){:}\mathsf{unit}\{\mathsf{a_S} \rightarrowtail \phi(n)\} \\
\quad \vdash \forall ns{:}\mathsf{list}\ A.\ \{\mathsf{a_S} \rightarrowtail \overline{\psi}(ns)\}\mathsf{iter}(ns){:}\mathsf{unit}\{\mathsf{a_S} \rightarrowtail \overline{\phi}(ns)\}
\end{array}$$

which assumes $f(n)$ changes the local auxiliary contribution from $\psi(n)$ to $\phi(n)$, relative to some ambient type A, PCM $(\mathbb{U}, \oplus, \mathbb{0})$, invariant I_*, and functions $\psi, \phi : A \to \mathbb{U}$. The overall effect of $\mathsf{iter}\ f\ ns$ is to change the local auxiliary contribution from $\overline{\psi}(ns)$ to $\overline{\phi}(ns)$, where $\overline{\psi}(ns)$ is $\bigoplus_{n \in ns} \psi(n)$ and similarly for $\overline{\phi}$.

By embedding SCSL into the type theory of the Calculus of Inductive Constructions (CiC) (Section 5), however, we can explicitly λ-abstract over f and give iter the specification (i.e., CiC type):

$$\begin{array}{l}
\mathsf{iter} : \forall f{:}(\forall n{:}A.\ \{\!|\mathsf{a_S} \rightarrowtail \psi(n)|\!\}\mathsf{unit}\{\!|\mathsf{a_S} \rightarrowtail \phi(n)|\!\}). \\
\qquad\qquad \forall ns{:}\mathsf{list}\ A.\ \{\!|\mathsf{a_S} \rightarrowtail \overline{\psi}(ns)|\!\}\mathsf{unit}\{\!|\mathsf{a_S} \rightarrowtail \overline{\phi}(ns)|\!\}
\end{array}$$

[8] We use pure lists and pattern-matching for clarity, although we can also prove the iterator for heap-allocated lists.

The proof outline for iter is:

$$
\mathsf{iter} \mathrel{\widehat{=}} \lambda f.\,\mathsf{fix\ loop}.\,ns.
\left(\begin{array}{l}
\mathsf{match}\ ns\ \mathsf{with} \\
\mid\ \mathsf{nil}\ \Rightarrow \{\mathsf{a_S} \rightarrowtail \overline{\psi}(\mathsf{nil})\}\ \mathsf{ret}\ ()\ \{\mathsf{a_S} \rightarrowtail \overline{\phi}(\mathsf{nil})\} \\
\mid\ n'{::}ns' \Rightarrow \\
\quad \left(\begin{array}{c}
\{\mathsf{a_S} \rightarrowtail \overline{\psi}(n'{::}ns')\} \\
\{\mathsf{a_S} \rightarrowtail (\psi(n') \oplus \overline{\psi}(ns'))\} \\
\{\mathsf{a_S} \rightarrowtail \psi(n') \circledast \mathsf{a_S} \rightarrowtail \overline{\psi}(ns')\} \\
\begin{array}{c|c}
\{\mathsf{a_S} \rightarrowtail \psi(n')\} & \{\mathsf{a_S} \rightarrowtail \overline{\psi}(ns')\} \\
f\ n' & \mathsf{loop}\ ns' \\
\{\mathsf{a_S} \rightarrowtail \phi(n')\} & \{\mathsf{a_S} \rightarrowtail \overline{\phi}(ns')\}
\end{array} \\
\{\mathsf{a_S} \rightarrowtail \phi(n') \circledast \mathsf{a_S} \rightarrowtail \overline{\phi}(ns')\} \\
\{\mathsf{a_S} \rightarrowtail (\phi(n') \oplus \overline{\phi}(ns'))\} \\
\{\mathsf{a_S} \rightarrowtail \overline{\phi}(n'{::}ns')\} \\
\mathsf{ret}\ () \\
\{\mathsf{a_S} \rightarrowtail \overline{\phi}(n'{::}ns')\}
\end{array}\right)
\end{array}\right)
$$

iter is defined *recursively* by fix with two arguments: loop for recursive calls and the list ns. The body pattern-matches the list: for nil, it returns unit and the overall effect preserves the local contribution $\overline{\psi}(\mathsf{nil}) = \mathbb{0} = \overline{\phi}(\mathsf{nil})$; otherwise for $n'{::}ns'$, it concurrently applies f to the head element n' and recurses on the tail ns', then returns unit, so the contribution changes from $\overline{\psi}(n'{::}ns')$ to $\overline{\phi}(n'{::}ns')$.

Defining xincr2 to be either the sincr2 $\mathrel{\widehat{=}} \lambda n.(\mathsf{incr}(n); \mathsf{incr}(n))$ or cincr2 $\mathrel{\widehat{=}} \lambda n.(\mathsf{incr}(n) \parallel \mathsf{incr}(n))$ has the same effect of incrementing by twice the argument (Section 2.2), independently of whether the program is sequential or concurrent:

$$\forall n{:}\mathbb{N}.\{\mathsf{a_S} \rightarrowtail 0\}\ \mathsf{xincr2}(n)\ \{\mathsf{a_S} \rightarrowtail 2 \times n\}$$

Then, we can show that iterating xincr2 increments by twice the sum of the list, using $\psi(\_) = 0$ and $\phi(n) = 2 \times n$:

$$\forall ns{:}\mathsf{list}\ \mathbb{N}.\{\mathsf{a_S} \rightarrowtail 0\}\ \mathsf{iter\ xincr2}\ ns\ \{\mathsf{a_S} \rightarrowtail 2 \times \textstyle\sum_{n \in ns} n\}$$

Finally, we can iterate again to add up a list of lists of numbers, using $\psi(\_) = 0$ and $\phi(ns) = \sum_{n \in ns} n$ for the outer iter:

$$\forall nss{:}\mathsf{list}\ (\mathsf{list}\ \mathbb{N}).\{\mathsf{a_S} \rightarrowtail 0\}\ \mathsf{iter}\ (\mathsf{iter\ xincr2})\ nss\ \{\mathsf{a_S} \rightarrowtail 2 \times \sum_{n \in ns \in nss} n\}$$

We can also iterate the concurrent set cadd procedure to show that if initially none of the elements of the list are in the set, then afterwards all of the elements are in the list, using $\psi(v) = \mathsf{out}(v)$ and $\phi(v) = \mathsf{in}(v)$:

$$\forall vs{:}\mathsf{list}\ V.\{\mathsf{a_S} \rightarrowtail \overline{\mathsf{out}}(vs)\}\ \mathsf{iter\ cadd}\ vs\ \{\mathsf{a_S} \rightarrowtail \overline{\mathsf{in}}(vs)\}$$

Although iter has a dynamic forking pattern that depends on the list argument and although f may spawn multiple threads, subjective auxiliary state gives the iterator a *robust* specification. We can use the *same* proof of iter for different types of list elements (e.g., $\mathbb{N}$, list $\mathbb{N}$, and V), different lists (e.g., any number of elements), and procedures with different internal concurrency (e.g., one thread in sincr2, two threads in cincr2, and argument-dependent in (iter xincr2)).

4. Language and Logic

Language In the tradition of axiomatic program logics, the language of SCSL splits into purely-functional *expressions* e (v when the expression is a value), and *commands* C with effects of divergence, state, and concurrency, and *procedures* F for commands with arguments. Expressions are classified by types A, which, for the purposes of this section, range over base types unit, nat, bool, and pointers ptr (isomorphic to nat), all with the usual values.

The syntax of commands and procedures is:

$$
\begin{array}{rl}
C ::= & \mathsf{lock} \mid \mathsf{unlock}_{\Phi} \mid x \leftarrow C_1; C_2 \mid C_1 \parallel C_2 \\
& \mid \mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid F(e) \\
F ::= & f \mid \mathsf{alloc} \mid \mathsf{dealloc} \mid \mathsf{read} \mid \mathsf{write} \mid \mathsf{ret} \mid \mathsf{fix}\, f.x.C
\end{array}
$$

Commands and procedures include memory and locking operations, a monadic unit ret v that returns v and terminates, a monadic bind (i.e., sequential composition) $x \leftarrow C_1;\ C_2$ runs C_1 then substitutes its result v_1 for x to run C_2 (we write $C_1; C_2$ when $x \notin \mathsf{FV}(C_2)$), fork-join parallel composition $C_1 \parallel C_2$, a conditional, a procedure application $F(e)$, a procedure variable f, or a fixed-point fix $f.x.C$ for recursive procedures. Note that memory operations and ret are technically procedures because they take arguments.

Like in a monadic functional language, such as Haskell, commands return values. This enables avoiding mutable variables (i.e., a stack) in favor of immutable variables, which can be substituted with expressions. A stack imposes a number of technical difficulties in a Hoare-style logic [2, 23, 24], especially for supporting procedures [14], so we prefer to avoid it.

In Section 5, we embed SCSL into the Calculus of Inductive Constructions (CiC), which immediately provides us with a pure language of expressions and specifications-level objects such as assertions and auxiliary functions. The embedding also gives us access to arbitrary CiC inductive types (e.g., list) and constructs for pattern-matching against them. We can also write procedures that abstract over *commands* and *procedures* (*cf.* the iterator in Section 3.2). Presenting the full syntax of CiC is beyond the scope of this paper, so we only present the fragment relevant to effectful programs and Hoare-style reasoning about them.

Specifications SCSL is parametrized in the auxiliary PCM $\mathbb{U}$ and a resource invariant I_* index by $\mathbb{U}$ that describes the single, global resource protected by the lock and unlock commands.

A command C satisfies the Hoare triple $\{p\}C : A\{q\}$ if it respects mutual exclusion and is memory-safe when executed from a world satisfying p, and concurrently with *any* environment that respects mutual exclusion. Furthermore, if C terminates, it returns a value of type A in a world satisfying q. q may use a dedicated variable res of type A to name the return result.

We use a *procedure triple*, $\forall x{:}B.\ \{p\}F\ (x) : A\{q\}$, to specify a potentially recursive procedure F taking an argument x of type B to a result of type A. The assertions p and q may depend on x. We use Cartesian products $A_1 \times A_2$ for functions with more than one argument, but curry the function for readability.

The SCSL judgments are *hypothetical* under a context Γ that maps *program variables* x to their type and *procedure variables* f to their specification. Each specification is allowed to depend on the variables declared to the left.

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

Γ does not bind logical variables. In first-order Hoare logics, logical variables are implicitly universally quantified with global scope. In SCSL, we limit their scope to the Hoare triple in which they appear. This is required for specifying recursive procedures, where a logical variable may be instantiated differently in each recursive call [14]. We also assume a formation requirement on Hoare triples $\mathsf{FLV}(p) \supseteq \mathsf{FLV}(q)$, i.e., that all free logical variables of the postcondition also appear in the precondition.

Figure 3 presents the inference rules for the SCSL judgments for Hoare triples $\Gamma \vdash \{p\}\ C : A\ \{q\}$ and $\Gamma \vdash \forall x : B.\{p\}\ F\ (x) : A\{q\}$. We have discussed the memory and locking commands in Section 2, so we proceed to describe the remaining inference rules.

Parallel composition The parallel composition $C_1 \parallel C_2$ forks into two children threads for C_1 and C_2, and when they terminate with return values v_i of type A_i, the threads are joined into a single return of the pair (v_1, v_2) of type $A_1 \times A_2$. The rule reads as follows: (1) the parallel composition precondition $p_1 \circledast p_2$ can be split into p_1 and p_2, (2) each thread C_i must be verified separately with precondition p_i and postcondition q_i, and (3) the postconditions q_1 and q_2 can be recombined into the parallel composition postcondi-

$$\begin{array}{llll}
\Gamma \vdash \forall x{:}A. & \{[\mu,\sigma,\omega]\,\mathsf{emp}\} & \mathsf{alloc}\ x : \mathsf{ptr} & \{[\mu,\sigma,\mathsf{if}\ \mu = \mathsf{Own}\ \mathsf{then}\ \omega\ \mathsf{else}\ -]\ \mathsf{res} \mapsto x\} \\
\Gamma \vdash \forall x{:}\mathsf{ptr}. & \{[\mu,\sigma,\omega]\ x \mapsto -\} & \mathsf{dealloc}\ x : \mathsf{unit} & \{[\mu,\sigma,\mathsf{if}\ \mu = \mathsf{Own}\ \mathsf{then}\ \omega\ \mathsf{else}\ -]\ \mathsf{emp}\} \\
\Gamma \vdash \forall x{:}\mathsf{ptr}. & \{[\mu,\sigma,\omega]\ x \mapsto \nu\} & \mathsf{read}\ x : A & \{[\mu,\sigma,\mathsf{if}\ \mu = \mathsf{Own}\ \mathsf{then}\ \omega\ \mathsf{else}\ -]\ x \mapsto \nu \wedge \mathsf{res} = \nu\} \\
\Gamma \vdash \forall x{:}\mathsf{ptr}.\forall y{:}A. & \{[\mu,\sigma,\omega]\ x \mapsto -\} & \mathsf{write}\ x\ y : \mathsf{unit} & \{[\mu,\sigma,\mathsf{if}\ \mu = \mathsf{Own}\ \mathsf{then}\ \omega\ \mathsf{else}\ -]\ x \mapsto y\} \\
\Gamma \vdash & \{[\cancel{\mathsf{Own}},\sigma,-]\ \mathsf{emp}\} & \mathsf{lock} : \mathsf{unit} & \{\exists\omega.[\mathsf{Own},\sigma,\omega]\ I_*\ (\sigma \oplus \omega)\} \\
\Gamma \vdash & \{[\mathsf{Own},\sigma,\omega]\ I_*\ (\Phi(\sigma) \oplus \omega)\} & \mathsf{unlock}_{\Phi} : \mathsf{unit} & \{[\cancel{\mathsf{Own}},\Phi(\sigma),-]\ \mathsf{emp}\} \\
\Gamma \vdash \forall x{:}A. & \{[\mu,\sigma,\omega]\ \mathsf{emp}\} & \mathsf{ret}\ x : A & \{[\mu,\sigma,\mathsf{if}\ \mu = \mathsf{Own}\ \mathsf{then}\ \omega\ \mathsf{else}\ -]\ \mathsf{emp} \wedge \mathsf{res} = x\}
\end{array}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash \{e = \mathsf{true} \wedge p\}C_1 : A\{q\} \\ \Gamma \vdash \{e = \mathsf{false} \wedge p\}C_2 : A\{q\}\end{array}}{\Gamma \vdash \{p\}\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : A\{q\}}\ \textsc{If} \qquad \dfrac{\Gamma \vdash \{p_1\}C : A\{q_1\} \qquad \Gamma \vdash \{p_2\}C : A\{q_2\}}{\Gamma \vdash \{p_1 \wedge p_2\}C : A\{q_1 \wedge q_2\}}\ \textsc{Conj} \qquad \dfrac{\Gamma \vdash \{p\}C : A\{q\} \qquad \alpha \notin \mathsf{dom}\ \Gamma}{\Gamma \vdash \{\exists\alpha{:}A.p\}C : A\{\exists\alpha{:}A.q\}}\ \textsc{Exist}$$

$$\dfrac{\Gamma \vdash \{p_1\}C : A\{q_1\} \qquad \Gamma \vdash (p_1, q_1) \sqsubseteq (p_2, q_2)}{\Gamma \vdash \{p_2\}C : A\{q_2\}}\ \textsc{Conseq} \qquad \dfrac{\Gamma \vdash \{p\}C : A\{q\}}{\Gamma \vdash \{p \circledast r\}C : A\{q \circledast r\}}\ \textsc{Frame}$$

$$\dfrac{\Gamma \vdash \{p\}C_1 : B\{q\} \qquad \Gamma, x{:}B \vdash \{[x/\mathsf{res}]q\}C_2 : A\{r\} \qquad x \notin \mathsf{FV}(r)}{\Gamma \vdash \{p\}x \leftarrow C_1; C_2 : A\{r\}}\ \textsc{Seq} \qquad \dfrac{\Gamma \vdash \{p_1\}C_1 : A_1\{q_1\} \qquad \Gamma \vdash \{p_2\}C_2 : A_2\{q_2\}}{\Gamma \vdash \{p_1 \circledast p_2\}C_1 \parallel C_2 : A_1 \times A_2\{[\mathsf{res}.1/\mathsf{res}]q_1 \circledast [\mathsf{res}.2/\mathsf{res}]q_2\}}\ \textsc{Par}$$

$$\dfrac{\forall x{:}B.\{p\}f(x) : A\{q\} \in \Gamma}{\Gamma \vdash \forall x{:}B.\{p\}f(x) : A\{q\}}\ \textsc{Hyp} \qquad \dfrac{\Gamma, \forall x{:}B.\{p\}f(x) : A\{q\}, x{:}B \vdash \{p\}C : A\{q\}}{\Gamma \vdash \forall x{:}B.\{p\}(\mathsf{fix}\,f.x.C)(x) : A\{q\}}\ \textsc{Fix} \qquad \dfrac{\Gamma \vdash \forall x{:}B.\{p\}F(x) : A\{q\} \qquad \Gamma \vdash e : B}{\Gamma \vdash \{[e/x]p\}F(e) : A\{[e/x]q\}}\ \textsc{App}$$

Figure 3. Subjective CSL rules $\Gamma \vdash \{p\}C : A\{q\}$ and $\Gamma \vdash \forall x{:}B.\{p\}F(x) : A\{q\}$.

tion $q_1 \circledast q_2$, with the projections from the resulting pair substituted for the dedicated variable res.

The precondition $p_1 \circledast p_2$ expresses that the initial world for the parent thread has the form $h_1 \oplus h_2; \mu_1 \oplus \mu_2; \sigma_1 \oplus \sigma_2; \omega$. The left child receives the world $h_1; \mu_1; \sigma_1; \sigma_2 \oplus \omega$ in its precondition p_1, and dually for the right child and p_2. The splitting of h, μ and σ components reflects that the children divide the local heap and contribution of the parent thread. The left child's environment contribution $\sigma_2 \oplus \omega$ reflects that upon forking C_2 becomes part of C_1's environment, and a dual remark applies to the right child.

The parallel composition rule thus states that if a partition satisfying the preconditions can be found of the initial world, then a partition satisfying both postconditions can be found of the ending world. Intuitively, the rule is sound because the required partition of the ending state can be obtained by executing C_1, C_2, and their common environment in an interleaved manner, while updating their respective values of $\mathsf{a_S}$ and $\mathsf{a_O}$. Whenever one thread's $\mathsf{a_S} \rightarrowtail \sigma_i$ is updated, the $\mathsf{a_O} \hookrightarrow \omega_j$'s of the other two threads must be updated correspondingly in order to preserve the coherence between views, that is, that every thread i sees the same total $\tau = \sigma_i \oplus \omega_i$.

Fixed points We support a combinator fix for general recursion, rather than just while-loops. The iterator (Section 3.2) requires fix because of its dynamic forking structure; it isn't tail recursive and thus difficult to write using a while-loop. The inference rule requires proving a Hoare triple for the procedure body, under a hypothesis that the recursive calls satisfy the same triple. In practice, when writing recursive programs, the assertions p and q have to be supplied by hand (they cannot be inferred automatically), and they essentially correspond to a loop invariant.

Framing The FRAME rule allows splitting the world into $p \circledast r$, verifying the command with precondition p and postcondition q, then recombining the postcondition with the framed world into $q \circledast r$. Note that the rule uses $\circledast$ which allows auxiliaries to also be framed. Somewhat surprisingly, the auxiliary PCM $\mathbb{U}$ need not be *cancellative*.[9] This allows us to prove a concurrent maximum program (not shown) that uses the PCM $(\mathbb{N}, \max, 0)$ which is *not* cancellative, yet we can frame the thread's auxiliary contribution.

Cancellativity for *heaps* is required in the soundness proofs for CSL (and SCSL) in order to uniquely determine the heap transfered between the thread and the shared resource upon locking and unlocking [19, 3]. In SCSL, the auxiliary PCM need not be cancellative, because there is no analogous transfer of auxiliaries.

Other Rules 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 Section 5, this judgment will correspond to 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 $p_2 \Rightarrow p_1$ for strengthening the precondition and $q_1 \Rightarrow q_2$ for weakening the postcondition, to deal with the local scope of logical variables (*cf.* Section 5.4). The other rules are standard from Hoare logic.

5. Semantics

We define semantic objects of *actions* for heap mutation and *action trees* for control flow, which yield finite, partial *approximations* of the behavior of SCSL commands; their *operational semantics* gives the low-level execution relative to the full heap (Section 5.1). We define the high-level notions of *coherent configuration* to maintain an explicit partition between the shared state heaplet and the thread's and environment's private heaplets and auxiliaries, and *subjective rely/guarantee* transitions between configurations for environment and thread interference that respect *mutual exclusion* (Section 5.2). Both the operational semantics and subjective rely/guarantee transitions use a large footprint, although SCSL enjoys small footprint specifications.

We relate the low-level operational semantics to the high-level subjective rely/guarantee transitions by a *modal* always predicate (Section 5.3) that ensures a tree is resilient to any amount of subjective rely interference, and that all operational steps by a tree are memory-safe and correspond to subjective guarantee interference.

The denotational semantics (Section 5.4) interprets judgments by the *monadic Hoare type* $\{\!|p|\!\}A\{\!|q|\!\}$, 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. 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.

[9] Cancellativity means that if $\alpha \oplus \beta = \beta \oplus \gamma$, then $\beta = \gamma$; a property that the heap PCM satisfies and is often associated with the FRAME rule.

$$\frac{\ell \notin \mathsf{dom} h}{h; \mathsf{Alloc}\ v \rightsquigarrow_{\mathsf{a}} [h|\ell \Mapsto v]; \ell} \qquad \frac{\ell \in \mathsf{dom} h}{h; \mathsf{Dealloc}\ \ell \rightsquigarrow_{\mathsf{a}} h \setminus \ell; ()}$$

$$\frac{h(\ell) = v : A}{h; \mathsf{Read}_A\ \ell \rightsquigarrow_{\mathsf{a}} h; v} \qquad \frac{\ell \in \mathsf{dom} h}{h; \mathsf{Write}\ \ell\ v \rightsquigarrow_{\mathsf{a}} [h|\ell \Mapsto v]; ()} \qquad \frac{}{h; \mathsf{Idle} \rightsquigarrow_{\mathsf{a}} h; ()}$$

$$\frac{h(\ell) = v_1}{h; \mathsf{CAS}\ \ell\ v_1\ v_2 \rightsquigarrow_{\mathsf{a}} [h|\ell \Mapsto v_2]; \mathsf{true}} \qquad \frac{h(\ell) = v \qquad v \neq v_1}{h; \mathsf{CAS}\ x\ v_1\ v_2 \rightsquigarrow_{\mathsf{a}} h; \mathsf{false}}$$

$$\frac{h; a \rightsquigarrow_{\mathsf{a}} h'; v}{h; \mathsf{Cons}\ a\ \Phi\ k \overset{\mathsf{nil}}{\rightsquigarrow}_{\mathsf{t}} h'; (k\ v)} \qquad \frac{}{h; \mathsf{Par}\ (\mathsf{Ret}\ v_1)\ (\mathsf{Ret}\ v_2)\ k \overset{\mathsf{nil}}{\rightsquigarrow}_{\mathsf{t}} h; k\ (v_1, v_2)}$$

$$\frac{h; t_1 \overset{\pi}{\rightsquigarrow}_{\mathsf{t}} h'; t_1'}{h; \mathsf{Par}\ t_1\ t_2\ k \overset{\mathsf{L}::\pi}{\rightsquigarrow}_{\mathsf{t}} h'; \mathsf{Par}\ t_1'\ t_2\ k} \qquad \frac{h; t_2 \overset{\pi}{\rightsquigarrow}_{\mathsf{t}} h'; t_2'}{h; \mathsf{Par}\ t_1\ t_2\ k \overset{\mathsf{R}::\pi}{\rightsquigarrow}_{\mathsf{t}} h'; \mathsf{Par}\ t_1\ t_2'\ k}$$

Figure 4. Action stepping $h; a \rightsquigarrow_{\mathsf{a}} h'; v$ (top) and tree stepping $h; t \overset{\pi}{\rightsquigarrow}_{\mathsf{t}} h'; t'$ (bottom).

The *soundness* of SCSL (Section 5.5) follows from showing that the denotation of memory operations satisfy the appropriate always predicate, and that always satisfies certain closure conditions. In Section 6, we discuss the relationship of our denotational semantics to existing semantics for CSL.

We choose the Calculus of Inductive Constructions (CiC) [18] as our meta logic. This has several important benefits. First, we can define a *shallow embedding* of SCSL into CiC that allows us to program and prove directly *with the semantic objects*, thus immediately lifting SCSL 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* [17, 18], specification-level functions (e.g., auxiliary functions Φ), and programming-level higher-order procedures (e.g., the iterator). Finally, we were able to mechanize the entire semantics and metatheory [16] in the Coq proof assistant implementation of CiC.

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 $x \in A$.

5.1 Actions, Trees, and Operational Semantics

The type family action A of A-returning **actions** includes constructors for the SCSL memory commands, an idle action returning unit, and a compare-and-swap CAS used to implement locking:

$$\begin{array}{llll}
\mathsf{Alloc}\ (v{:}B) & : \mathsf{action\ ptr} & \mathsf{Read}_A\ (x{:}\mathsf{ptr}) & : \mathsf{action}\ A \\
\mathsf{Dealloc}\ (x{:}\mathsf{ptr}) & : \mathsf{action\ unit} & \mathsf{Write}\ (x{:}\mathsf{ptr})\ (v{:}B) & : \mathsf{action\ unit} \\
\mathsf{Idle} & & : \mathsf{action\ unit} & \\
\mathsf{CAS}\ (x{:}\mathsf{ptr})\ (v_1{:}B)\ (v_2{:}B) & : \mathsf{action\ bool} & &
\end{array}$$

The operational semantics of actions $h; a \rightsquigarrow_{\mathsf{a}} h'; v$ (Figure 4, top) steps an A-returning action a relative to an initial heap h, and produces an ending heap h' and result v of type A. The operational semantics of the basic memory operations is standard. The CAS action takes a pointer ℓ and two values v_1 and v_2, it atomically checks whether ℓ contains v_1 and, if so, replaces it with v_2 and returns true; otherwise, it preserves the heap and returns false.

The type family tree A of A-returning **trees** is defined by an *iterated inductive definition*:

$$\begin{array}{ll}
\mathsf{Bottom} & : \mathsf{tree}\ A \\
\mathsf{Ret}\ (v{:}A) & : \mathsf{tree}\ A \\
\mathsf{Cons}\ (a{:}\mathsf{action}\ B)\ (\Phi{:}\mathbb{U} \rightharpoonup_{\mathsf{L}} \mathbb{U})(k{:}B \to \mathsf{tree}\ A) & : \mathsf{tree}\ A \\
\mathsf{Par}\ (t_1{:}\mathsf{tree}\ B_1)\ (t_2{:}\mathsf{tree}\ B_2)\ (k{:}B_1 \times B_2 \to \mathsf{tree}\ A) & : \mathsf{tree}\ A
\end{array}$$

Since trees have finite depth, they can only *approximate* divergent computations, thus the Bottom tree indicates an incomplete approximation. Ret v is a terminal computation that returns value $v : A$. Cons $a\ \Phi\ k$ sequentially composes a B-returning action a with a continuation k that takes a's return value and generates the rest of the approximation; the tree also carries a local auxiliary function $\Phi{:}\mathbb{U} \rightharpoonup_{\mathsf{L}} \mathbb{U}$ for mutating auxiliary state (Section 5.2). 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.

To explicitly quantify over the nondeterminism of concurrent interleaving, we define a schedule ζ to be a list of paths, where a path π is a list of L or R symbols that traverse the Left or Right branches of Par trees and select a β-redex (i.e., an action or the parallel composition of two Ret trees).

The small-step operational semantics of trees $h; t \overset{\pi}{\rightsquigarrow}_{\mathsf{t}} h', t'$ (Figure 4, bottom) is defined inductively on π to step tree t from initial heap h to a reduced tree t' in ending heap h'. Stepping is undefined for the Bottom and Ret trees. For the Cons and Par trees, the path π selects a β-redex and performs the appropriate reduction. Note that only the heap and action in the tree are needed to define stepping, which captures the intuition that auxiliary state $\mathbb{U}$ and auxiliary functions Φ have no bearing on the operational semantics.

The denotation of SCSL Hoare triples will establish that "well-specified programs don't go wrong". We follow a Milner-style approach where only well-typed (i.e., well-specified) programs are given meaning and ensure that every action and tree encountered in the operational semantics is memory-safe relative to the given heap; thus we can avoid explicit fault states in the operational semantics and defining the semantics of unsafe programs.

Definition 2 (Action and auxiliary function at a path)**.** *Given a tree t and path π, the action a and local auxiliary function Φ appear in t at path π, written $(a, \Phi) = t \mathbin{@} \pi$, iff*

$$\begin{array}{lll}
\pi = \mathsf{nil} & \textit{and}\ t = \mathsf{Cons}\ a\ \Phi\ k & \\
\textit{or}\ \pi = \mathsf{nil} & \textit{and}\ t = \mathsf{Par}\ (\mathsf{Ret}\ \_)\ (\mathsf{Ret}\ \_)\ k\ \textit{and}\ (a, \Phi) = (\mathsf{Idle}, \mathsf{id}) & \\
\textit{or}\ \pi = \mathsf{L}::\pi' & \textit{and}\ t = \mathsf{Par}\ t_1\ \_\ \_ & \textit{and}\ (a, \Phi) = t_1 \mathbin{@} \pi' \\
\textit{or}\ \pi = \mathsf{R}::\pi' & \textit{and}\ t = \mathsf{Par}\ \_\ t_2\ \_ & \textit{and}\ (a, \Phi) = t_2 \mathbin{@} \pi'
\end{array}$$

We write $a = t \mathbin{@} \pi$ instead of $(a, -) = t \mathbin{@} \pi$ when we are interested only in the action a, and likewise for Φ.

Definition 3 (Safety)**.** *1. Action a is safe for the heap h, if there exist h' and v such that $h; a \rightsquigarrow_{\mathsf{a}} h', v$.*

2. Tree t is safe for a heap h and path π, if $a = t@\pi$ implies that a is safe for h.

3. Tree t is safe for a heap h and a schedule ζ if either (1) $\zeta = \mathsf{nil}$, or (2) if $\zeta = \pi::\zeta'$ then t is safe for h and π, and t' is safe for h' and ζ' for all h' and t' such that $h; t \overset{\pi}{\rightsquigarrow}_{\mathsf{t}} h'; t'$.

4. Tree t is safe for a heap h, written memsafe $h\ t$, *if it is safe for h, for any schedule ζ.*

Intuitively, t is memory-safe for heap h if no matter which steps one takes through t starting from h, the actions in the tree will not attempt to dereference a non-existent or ill-typed memory cell, or deallocate an unallocated cell. Bottom and Ret trees are trivially safe because they don't step. A Cons tree is safe if it's head action is safe and the continuation is safe. A Par tree is safe if both of it's subtrees are safe, and when they return the continuation is safe.

5.2 Coherent Configurations and Subjective Rely/Guarantee

The operational semantics yields execution over the full heap, but doesn't contain enough information for enforcing SCSL specifications regarding mutual exclusion for critical sections over the resource invariant and auxiliaries. We need to extend heaps with additional information which is the *logical state* of SCSL.

We define *coherent configurations* $(h_{\mathsf{S}}; \mu_{\mathsf{S}}; \sigma_{\mathsf{S}} \mid h_{\mathsf{R}} \mid h_{\mathsf{O}}; \mu_{\mathsf{O}}; \sigma_{\mathsf{O}})$ consisting of the thread's private heap h_{S}, lock ownership status μ_{S}, and auxiliary σ_{S}; the shared heap h_{R} containing the dedicated lock location lk and the shared resource satisfying the resource invariant I_*; and the environment's private state h_{O}, μ_{O}, and σ_{O}. Subjectivity thus appears in the semantics as the logical partition of heaps, lock ownership status, and auxiliaries from the point of view of the thread being specified. Although environment threads have their own private states, the configuration conflates them into a single private state for the environment as a whole.

Coherent configurations are defined relative to the user-chosen auxiliary PCM $\mathbb{U}$ and resource invariant I_* over $\mathbb{U}$ and heap such that $I_*\,(\alpha)$ is *precise*[10] for all $\alpha \in \mathbb{U}$. As in CSL, precision is required to uniquely determine the invariant heaplet to be transferred between the shared and private heaplets. Moreover, we assume a reserved location lk for implementing the lock.

Definition 4 (Coherent Configuration)**.** *A* coherent configuration *is a 7-tuple* $c = (h_{\mathsf{S}}; \mu_{\mathsf{O}}; \sigma_{\mathsf{O}} \mid h_{\mathsf{R}} \mid h_{\mathsf{O}}; \mu_{\mathsf{O}}; \sigma_{\mathsf{O}})$ *such that:*

1. $\mathsf{valid}\,(h_{\mathsf{S}} \oplus h_{\mathsf{R}} \oplus h_{\mathsf{O}})$, $\mathsf{valid}\,(\mu_{\mathsf{S}} \oplus \mu_{\mathsf{O}})$, $\mathsf{valid}\,(\sigma_{\mathsf{S}} \oplus \sigma_{\mathsf{O}})$
2. $\mu_{\mathsf{S}} \oplus \mu_{\mathsf{O}} = \cancel{\mathsf{Own}} \Rightarrow h_{\mathsf{R}} \models_{\mathsf{SL}} \mathsf{lk} \mapsto \mathsf{false} * I_*\,(\sigma_{\mathsf{S}} \oplus \sigma_{\mathsf{O}})$
3. $\mu_{\mathsf{S}} \oplus \mu_{\mathsf{O}} = \mathsf{Own} \Rightarrow h_{\mathsf{R}} \models_{\mathsf{SL}} \mathsf{lk} \mapsto \mathsf{true}$

Intuitively, all components of a configuration join in a valid way (1). Mutual exclusion means that (2) when nobody owns lk ($\mu_{\mathsf{S}} \oplus \mu_{\mathsf{O}} = \cancel{\mathsf{Own}}$), then the shared heap contains the lock lk set to false and a heaplet satisfying $I_*(\sigma_{\mathsf{S}} \oplus \sigma_{\mathsf{O}})$ relative to the total auxiliary; and (3) if either the thread or its environment owns lk ($\mu_{\mathsf{S}} \oplus \mu_{\mathsf{O}} = \mathsf{Own}$), then the shared heap only contains the lock lk set to true because the owner has acquired the invariant heaplet.

The large footprint configuration is needed to define the transitions and establish the soundness of the logic. However, the operational semantics only uses the full heap $h_{\mathsf{S}} \oplus h_{\mathsf{R}} \oplus h_{\mathsf{O}}$, and the assertions only require the small footprint of the world $h_{\mathsf{S}}; \mu_{\mathsf{S}}; \sigma_{\mathsf{S}}; \sigma_{\mathsf{O}}$. Recall that the environment's auxiliary σ_{O} is needed in the critical section to establish its invariance and prove that I_* is preserved (*cf.* the proof of incr in Section 2.4); but the environment's components h_{O} and μ_{O} are omitted from assertions because they are not needed for local verification of a thread.

Definition 5. *A configuration* $c = (h_{\mathsf{S}}; \mu_{\mathsf{S}}; \sigma_{\mathsf{S}} \mid h_{\mathsf{R}} \mid h_{\mathsf{O}}; \mu_{\mathsf{O}}; \sigma_{\mathsf{O}})$ *has heap* $\mathsf{hp}(c) \mathrel{\widehat{=}} h_{\mathsf{S}} \oplus h_{\mathsf{R}} \oplus h_{\mathsf{O}}$ *and world* $\mathsf{wd}(c) \mathrel{\widehat{=}} h_{\mathsf{S}}; \mu_{\mathsf{S}}; \sigma_{\mathsf{S}}; \sigma_{\mathsf{O}}$.

Given the spatial description of configurations, we can define the temporal notions of *subjective rely/guarantee* transitions $\rightsquigarrow^*_{\mathsf{R}}$ and $\overset{\Phi}{\rightsquigarrow}_{\mathsf{G}}$ for environment and thread interference on a configuration that respect mutual exclusion and auxiliary updates.

Definition 6 (Subjective guarantee)**.** *The* subjective guarantee *transition* $c \overset{\Phi}{\rightsquigarrow}_{\mathsf{G}} c'$ *holds relative to* $\Phi : \mathbb{U} \rightharpoonup_{\mathsf{L}} \mathbb{U}$ *if the configurations* c *and* c' *have one of the forms (where* $d_{\mathsf{O}} = h_{\mathsf{O}}; \mu_{\mathsf{O}}; \sigma_{\mathsf{O}}$*):*

$$
\begin{array}{llllll}
\mathsf{Priv} & c\; = (h_{\mathsf{S}} & ; \mu_{\mathsf{S}} & ; \sigma_{\mathsf{S}} & \mid h_{\mathsf{R}} & \mid d_{\mathsf{O}}) \\
 & c' = (h'_{\mathsf{S}} & ; \mu_{\mathsf{S}} & ; \sigma_{\mathsf{S}} & \mid h_{\mathsf{R}} & \mid d_{\mathsf{O}}) \\
\hline
\mathsf{Acq} & c\; = (h_{\mathsf{S}} & ; \cancel{\mathsf{Own}} & ; \sigma_{\mathsf{S}} & \mid [\mathsf{lk} \Mapsto \mathsf{false}] \oplus h_{\mathsf{inv}} & \mid d_{\mathsf{O}}) \\
 & c' = (h_{\mathsf{S}} \oplus h_{\mathsf{inv}} & ; \mathsf{Own} & ; \sigma_{\mathsf{S}} & \mid [\mathsf{lk} \Mapsto \mathsf{true}] & \mid d_{\mathsf{O}}) \\
\hline
\mathsf{Rel} & c\; = (h_{\mathsf{S}} \oplus h_{\mathsf{inv}} & ; \mathsf{Own} & ; \sigma_{\mathsf{S}} & \mid [\mathsf{lk} \Mapsto \mathsf{true}] & \mid d_{\mathsf{O}}) \\
 & c' = (h_{\mathsf{S}} & ; \cancel{\mathsf{Own}} & ; \Phi(\sigma_{\mathsf{S}}) & \mid [\mathsf{lk} \Mapsto \mathsf{false}] \oplus h_{\mathsf{inv}} & \mid d_{\mathsf{O}})
\end{array}
$$

Subjective guarantee transitions correspond to the changes that a thread can perform: mutation on the thread's Private heap, Acquiring the resource by locking and transferring the invariant heaplet from the shared state to the thread's private heap, and dually Releasing the resource. In the last case, the auxiliary function Φ uniquely determines how to change the thread's view σ_{S}. Note that in any case, the environment's private state is preserved.

The environment threads make complementary transitions on the shared state and the environment's private state. However, as we will define Hoare triples to abstract over arbitrary environment interference, a coarser view suffices to describe mutual exclusion.

Definition 7 (Subjective rely)**.** *The* subjective rely *transition* $c \rightsquigarrow^*_{\mathsf{R}} c'$ *holds if the configurations have the form* $c = (h_{\mathsf{S}}; \mu_{\mathsf{S}}; \sigma_{\mathsf{S}} \mid h_{\mathsf{R}} \mid h_{\mathsf{O}}; \mu_{\mathsf{O}}; \sigma_{\mathsf{O}})$ *and* $c' = (h_{\mathsf{S}}; \mu_{\mathsf{S}}; \sigma_{\mathsf{S}} \mid h'_{\mathsf{R}} \mid h'_{\mathsf{O}}; \mu'_{\mathsf{O}}; \sigma'_{\mathsf{O}})$*, and if* $\mu_{\mathsf{S}} = \mathsf{Own}$ *then* $\sigma'_{\mathsf{O}} = \sigma_{\mathsf{O}}$.

A subjective rely transition represents arbitrary environment interference that respects mutual exclusion. The environment can mutate its private state and the shared state arbitrarily as long as coherence is preserved. If the thread Owns the lock, then the environment's auxiliary must remain fixed. Intuitively, $\rightsquigarrow^*_{\mathsf{R}}$ is the *reflexive-transitive* closure of the *transpose* of $\rightsquigarrow_{\mathsf{G}}$, i.e., where mutation is done on the environment's private state instead of the thread's private state, and with any possible auxiliary function Φ. Note that in any case, the thread's private state is preserved.

5.3 Modal Predicates

We next define a number of *modal* predicates over *all* possible steps of execution to relate the operational semantics over heaps and the subjective rely/guarantee transitions over coherent configurations.

Definition 8 (Modal predicates)**.** *The predicates* $\mathsf{always}^{\zeta}\ c\ t\ p$, $\mathsf{always}\ c\ t\ P$*, and* $\mathsf{after}\ c\ t\ Q$ *are defined relative to a schedule* ζ*, configuration* c*,* A*-returning tree* t*, and predicates* $P : \mathsf{config} \to \mathsf{tree}\ A \to \mathsf{prop}$ *and* $Q : \mathsf{config} \to A \to \mathsf{prop}$*:*

$$
\begin{aligned}
&\mathsf{always}^{\zeta}\ c\ t\ P \mathrel{\widehat{=}} \\
&\quad \forall c_{\mathsf{R}}.\ c \rightsquigarrow^*_{\mathsf{R}} c_{\mathsf{R}} \Rightarrow \\
&\qquad \left(\begin{array}{l}
\mathsf{memsafe}\,(\mathsf{hp}(c_{\mathsf{R}}))\ t \wedge P\ c_{\mathsf{R}}\ t \wedge \\
\forall \pi\ \zeta'\ h_{\mathsf{G}}\ t'.\ \zeta = \pi{::}\zeta' \ \wedge\ \mathsf{hp}(c_{\mathsf{R}}); t \overset{\pi}{\rightsquigarrow}_{\mathsf{t}} h_{\mathsf{G}}; t' \Rightarrow \\
\qquad\qquad \left(\begin{array}{r}
\exists c_{\mathsf{G}}\ \Phi.\ \mathsf{hp}(c_{\mathsf{G}}) = h_{\mathsf{G}} \wedge \Phi = t \mathbin{@} \pi \wedge \\
c_{\mathsf{R}} \overset{\Phi}{\rightsquigarrow}_{\mathsf{G}}\ c_{\mathsf{G}} \wedge \mathsf{always}^{\zeta'}\ c_{\mathsf{G}}\ t'\ P
\end{array} \right)
\end{array} \right) \\
&\mathsf{always}\ c\ t\ P \mathrel{\widehat{=}} \forall \zeta.\ \mathsf{always}^{\zeta}\ c\ t\ P \\
&\mathsf{after}\ c\ t\ Q \mathrel{\widehat{=}} \mathsf{always}\ c\ t\ (\lambda c'\ t'.\ \forall v'.\ t' = \mathsf{Ret}\ v' \Rightarrow Q\ c'\ v')
\end{aligned}
$$

$\mathsf{always}\ c\ t\ P$ expresses the fact that starting from configuration c, 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 interference. The helper predicate $\mathsf{always}^{\zeta}\ c\ t\ P$ is defined by induction on ζ: the environment is allowed to make arbitrary subjective rely interference from c to c_{R}, the resulting configuration must have a heap that's memory safe for t and the predicate $P\ c_{\mathsf{R}}\ t$ holds; moreover, if the schedule is $\pi{::}\zeta'$ and t steps to h_{G} and t', then there must be a subjective guarantee transition from c_{R} to c_{G} whose heap is h_{G}, and the predicate recurses on ζ', c_{G}, and t'. Mutual exclusion is thus ensured because all transitions conform to subjective rely/guarantee transitions.

$\mathsf{after}\ c\ t\ Q$ encodes that t is memory safe and respects mutual exclusion; however, $Q\ c'\ v'$ only holds if t steps completely to $\mathsf{Ret}\ v'$ in configuration c'.

[10] A heap predicate p is *precise* if it determines a unique subheap: for every heap h with subheaps h_1, h_2, if $h_1 \models_{\mathsf{SL}} p$ and $h_2 \models_{\mathsf{SL}} p$ then $h_1 = h_2$.

5.4 SCSL Denotational Semantics

The denotational semantics of SCSL commands and judgments are defined simultaneously. The mutual recursion is neccessary 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 SCSL programs as *sets* T of trees of increasing precision including the Bottom tree, which is the coarsest possible approximation of any program:

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

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

Definition 9 (Hoare types)**.** *Fix precondition* $p : \mathsf{world} \to \mathsf{prop}$ *postcondition* $q : A \to \mathsf{world} \to \mathsf{prop}$*, with free logical variables* $\mathsf{FLV}(p, q)$*. The Hoare type* $\{\!|p|\!\}A\{\!|q|\!\}$ *is defined as follows.*

$$\{\!|p|\!\}A\{\!|q|\!\} \mathrel{\widehat{=}} \{T \in \mathsf{prog}\ A \mid \forall \mathsf{FLV}(p,q)\ (c\,{:}\mathsf{config})\ (t \in T).$$
$$(\mathsf{wd}(c) \models p) \Rightarrow \mathsf{after}\ c\ t\ (\lambda c'\ v'.\, \mathsf{wd}(c') \models q\ v')\}$$

Intuitively, the denotation of a SCSL judgment $\{p\}C{:}A\{q\}$ is the set of trees T denoting the command C, together with a proof that for any initial configuration c whose world $\mathsf{wd}(c)$ satisfies the precondition p, then after executing any tree $t \in T$ from c produces some result value v' and final configuration c' whose world $\mathsf{wd}(c')$ satisfies postcondition q. The definition quantifies over the free logical variables of p and q in order to give these variables local scope, as stipulated in Section 4.

SCSL assertions (Section 2.3) are arbitrary CiC predicates of type $\mathsf{world} \to \mathsf{prop}$ over worlds, where $\mathsf{world} \mathrel{\widehat{=}} \mathsf{heap} \times \mathsf{mtx} \times \mathbb{U} \times \mathbb{U}$. For example, the predicate $\mathsf{a_S} \rightarrowtail \sigma'$ is defined as $\lambda w{:}\mathsf{world}.\,(\exists h\ \mu\ \omega.\, w = h; \mu; \sigma'; \omega)$. We retain the $w \models p$ notation for the application $p\ w$.

Lemma 1. *The type* $\{\!|p|\!\}A\{\!|q|\!\}$ *is a complete lattice, with set union as the join operator, and* $\{\mathsf{Bottom}\}$ *as the unit element.*

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

The **denotation of judgments** $\llbracket \Gamma \vdash J \rrbracket$ (Figure 5, top) turns SCSL judgments into CiC typing judgments ($\vdash_{\mathsf{CiC}}$). A command specification $\{p\} - {:}A\{q\}$ is denoted by the CiC type $\{\!|p|\!\}A\{\!|q|\!\}$, and a procedure specification $\forall x{:}B.\ \{p\} - {:}A\{q\}$ is denoted by the CiC monadic function type $\forall x{:}B.\ \{\!|p|\!\}A\{\!|q|\!\}$.

The $(p_1, q_1) \sqsubseteq (p_2, q_2)$ judgment generalizes the usual side conditions on the rule of CONSEQuence: (1) $p_2 \Rightarrow p_1$ for strengthening the precondition and (2) $q_1 \Rightarrow q_2$ for weakening the postcondition, and adapts them to the local interpretation of logical variables. The first conjunct in the denotation existentially quantifies the respective logic variables of p_2 and of p_1 to generalize (1); the quantification is existential to match the definition of Hoare types, which quantify universally over $\mathsf{FLV}(p, q)$ and have a negative occurrence of the precondition. The second conjunct generalizes (2) by universally quantifying the respective logic variables, taking into account that the logic variables of q_i are constrained by the precondition p_i.

The **denotation of commands and procedures** (Figure 5, bottom) 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 $\mathsf{prog}\ A$, and an A-returning procedure F with argument B is denoted by a set of trees in $B \to \mathsf{prog}\ A$.

$$\begin{aligned}
&\llbracket \cdot \rrbracket && \mathrel{\widehat{=}}\ \cdot \\
&\llbracket \Gamma, x{:}A \rrbracket && \mathrel{\widehat{=}}\ \llbracket \Gamma \rrbracket, x : A \\
&\llbracket \Gamma, \forall x{:}B.\ \{p\}\, f(x){:}A\, \{q\} \rrbracket && \mathrel{\widehat{=}}\ \llbracket \Gamma \rrbracket, f : \forall x{:}B.\ \{\!|p|\!\}A\{\!|q|\!\} \\[1ex]
&\llbracket \Gamma \vdash \{p\}\ C : A\, \{q\} \rrbracket && \mathrel{\widehat{=}}\ \llbracket \Gamma \rrbracket \vdash_{\mathsf{CiC}} \llbracket C \rrbracket : \{\!|p|\!\}A\{\!|q|\!\} \\
&\llbracket \Gamma \vdash \forall x{:}B. \{p\} F(x) : A\{q\} \rrbracket && \mathrel{\widehat{=}}\ \llbracket \Gamma \rrbracket \vdash_{\mathsf{CiC}} \llbracket F \rrbracket : \forall x{:}B.\ \{\!|p|\!\}A\{\!|q|\!\} \\
&\llbracket \Gamma \vdash e : A \rrbracket && \mathrel{\widehat{=}}\ \llbracket \Gamma \rrbracket \vdash_{\mathsf{CiC}} e : A \\
&\llbracket \Gamma \vdash (p_1, q_1) \sqsubseteq (p_2, q_2) \rrbracket && \mathrel{\widehat{=}} \\
&\quad \llbracket \Gamma \rrbracket \vdash_{\mathsf{CiC}} \forall w\ w'{:}\mathsf{world}.\ (w \models (\exists \bar{v}_2.\, p_2) \Rightarrow (\exists \bar{v}_1.\, p_1)) \wedge \\
&\qquad ((\forall \bar{v}_1.\, w \models p_1 \Rightarrow w' \models q_1) \Rightarrow (\forall \bar{v}_2.\, w \models p_2 \Rightarrow w' \models q_2)) \\
&\quad \text{where } \bar{v}_i \mathrel{\widehat{=}} \mathsf{FLV}(p_i, q_i)
\end{aligned}$$

$$\begin{aligned}
\llbracket \mathsf{alloc}_A \rrbracket\ (x{:}A) &\mathrel{\widehat{=}} \{\mathsf{Bottom}, \mathsf{Cons}\ (\mathsf{Alloc}\ x)\ \mathsf{id}\ \mathsf{Ret}\} \\
\llbracket \mathsf{dealloc} \rrbracket\ (x{:}\mathsf{ptr}) &\mathrel{\widehat{=}} \{\mathsf{Bottom}, \mathsf{Cons}\ (\mathsf{Dealloc}\ x)\ \mathsf{id}\ \mathsf{Ret}\} \\
\llbracket \mathsf{read}_A \rrbracket\ (x{:}\mathsf{ptr}) &\mathrel{\widehat{=}} \{\mathsf{Bottom}, \mathsf{Cons}\ (\mathsf{Read}_A\ x)\ \mathsf{id}\ \mathsf{Ret}\} \\
\llbracket \mathsf{write}_A \rrbracket\ (x{:}\mathsf{ptr})\ (y{:}A) &\mathrel{\widehat{=}} \{\mathsf{Bottom}, \mathsf{Cons}\ (\mathsf{Write}\ x\ y)\ \mathsf{id}\ \mathsf{Ret}\} \\
\llbracket \mathsf{lock} \rrbracket &\mathrel{\widehat{=}} \{t_k \mid k \in \mathbb{N}\} \\
\text{where}\quad t_0 &\mathrel{\widehat{=}} \mathsf{Bottom}, \\
t_{k+1} &\mathrel{\widehat{=}} \mathsf{Cons}\ (\mathsf{CAS}\ \mathsf{lk}\ \mathsf{false}\ \mathsf{true})\ \mathsf{id} \\
&\qquad\quad (\lambda x.\, \mathsf{if}\ x\ \mathsf{then}\ \mathsf{Ret}\ ()\ \mathsf{else}\ t_k) \\
\llbracket \mathsf{unlock}_\Phi \rrbracket &\mathrel{\widehat{=}} \{\mathsf{Bottom}, \mathsf{Cons}\ (\mathsf{Write}\ \mathsf{lk}\ \mathsf{false})\ \Phi\ \mathsf{Ret}\} \\
\llbracket \mathsf{ret}_A \rrbracket\ (x{:}A) &\mathrel{\widehat{=}} \{\mathsf{Bottom}, \mathsf{Ret}\ x\} \\
\llbracket x \leftarrow C_1; C_2 \rrbracket &\mathrel{\widehat{=}} \llbracket C_1 \rrbracket; (\lambda x.\ \llbracket C_2 \rrbracket) \\
\llbracket C_1 \parallel C_2 \rrbracket &\mathrel{\widehat{=}} \llbracket C_1 \rrbracket \parallel \llbracket C_2 \rrbracket \\
\llbracket \mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \rrbracket &\mathrel{\widehat{=}} \mathsf{if}\ e\ \mathsf{then}\ \llbracket C_1 \rrbracket\ \mathsf{else}\ \llbracket C_2 \rrbracket \\
\llbracket F\ (e) \rrbracket &\mathrel{\widehat{=}} \llbracket F \rrbracket\ (e) \\
\llbracket \mathsf{fix}\ f^{B,A,p,q}.\, x.\ C \rrbracket &\mathrel{\widehat{=}} \mathsf{lfp}_{\forall x:B.\ \{\!|p|\!\}A\{\!|q|\!\}}\ (\lambda f. \lambda x.\ \llbracket C \rrbracket)
\end{aligned}$$

Figure 5. Denotational semantics of SCSL judgments (top), and commands and procedures (bottom).

Each SCSL memory command is denoted by a pair of approximations: Bottom, and a Cons tree with the appropriate action and a Ret continuation. We use the identity function on $\mathbb{U}$ as our auxiliary function (which is obviously local) because memory commands don't change auxiliary state.

The lock command is denoted by the set of approximations of a loop around a CAS to atomically change the contents of the dedicated lock lk from false to true, until it succeeds. Intuitively, this is an explicit least fixed point construction. The unlock_Φ command writes false into the dedicated pointer lk and updates the auxiliary state according to the user-provided local auxiliary function Φ.

The sequential composition of commands uses the sequential composition $(T_1; K_2) \in \mathsf{prog}\ B$ of a set of trees $T_1 \in \mathsf{prog}\ A$ and a function into a set of trees $K_2 \in A \to \mathsf{prog}\ B$, which concatenates $t_1 \bowtie K_2$ each tree $t_1 \in T_1$ with a continuation in K_2.

$$\begin{aligned}
T_1; K_2 &\mathrel{\widehat{=}} \textstyle\bigcup \{t_1 \bowtie K_2 \mid t_1 \in T_1\} \\
\text{where}\qquad \mathsf{Ret}\ v \bowtie K &\mathrel{\widehat{=}} K\ v \\
\mathsf{Par}\ t_1\ t_2\ k \bowtie K &\mathrel{\widehat{=}} \{\mathsf{Par}\ t_1\ t_2\ k' \mid \forall x.\, k'\ x \in (k\ x \bowtie K)\} \\
\mathsf{Cons}\ a\ \Phi\ k \bowtie K &\mathrel{\widehat{=}} \{\mathsf{Cons}\ a\ \Phi\ k' \mid \forall x.\, k'\ x \in (k\ x \bowtie K)\} \\
\mathsf{Bottom} \bowtie K &\mathrel{\widehat{=}} \{\mathsf{Bottom}\}
\end{aligned}$$

The definition is well-founded because the continuation k' approximates trees in $k\ x \bowtie K$ for every x, as permitted by the iterated inductive definition.

The parallel composition of commands uses the parallel composition $T_1 \parallel T_2 \in \mathsf{prog}\ (A_1 \times A_2)$ of sets of trees $T_1 \in \mathsf{prog}\ A_1$ and $T_2 \in \mathsf{prog}\ A_2$, which includes Bottom and all the pairwise parallel compositions of trees in T_1 and T_2.

$$T_1 \parallel T_2 \mathrel{\widehat{=}} \{\mathsf{Bottom}\} \cup \{\mathsf{Par}\ t_1\ t_2\ \mathsf{Ret} \mid t_1 \in T_1 \wedge t_2 \in T_2\}$$

Since all SCSL program constructors preserve *monotonicity*, the $\mathsf{fix}\, f.x.C$ procedure can take the least fixed point lfp of the function $\lambda f.\, \lambda x.\ \llbracket C \rrbracket$ by the Knaster-Tarski theorem.

5.5 Modal Lemmas and Soundness Theorem

We culminate with the proof of soundness of the interpretation. We have carried all of these proofs in Coq [16], they usually proceed by an induction on the schedule ζ.

The Determinacy lemma means subjective guarantee transitions from a configuration that result in equal heaps, also result in equal configurations (i.e., with equal auxiliaries and heap partitions). Thus, stepping a tree in always uniquely determines the auxiliary state, which is crucial for the soundness of the CONJunction rule.

Lemma 2 (Determinacy of Subjective Guarantee transitions)**.** *If* $c \overset{\Phi}{\rightsquigarrow}_{\mathsf{G}} c_1$, $c \overset{\Phi}{\rightsquigarrow}_{\mathsf{G}} c_2$, *and* $\mathsf{hp}(c_1) = \mathsf{hp}(c_2)$, *then* $c_1 = c_2$.

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

Lemma 3 (Universal)**.** *If* $\mathsf{always}\ c\ t\ (\lambda c'\ t'.\, \mathsf{True})$, *then* always *commutes with universal quantification:*

$$\mathsf{always}\ c\, t\, (\lambda c'\, t'.\, \forall x.\, P\, x\, c'\, t') \text{ iff } \forall x.\, \mathsf{always}\ c\, t\, (\lambda c'\, t'.\, P\, x\, c'\, t').$$

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

Lemma 4 (Normality for after)**.** *If* $\mathsf{after}\ c\ t\ Q_1$ *and* $Q_1\ c'\ v' \Rightarrow Q_2\ c'\ v'$ *for all* c' *and* v', *then* $\mathsf{after}\ c\ t\ Q_2$.

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

Lemma 5 (Closure under sequential composition)**.** *If* $t_{12} \in t_1 \bowtie T_2$ *and* $\mathsf{after}\ c\ t_1\ (\lambda c'\, v'.\, \forall t'.\, t' \in T_2\ v' \Rightarrow \mathsf{after}\ c'\ t'\ Q)$, *then* $\mathsf{after}\ c\ t_{12}\ Q$.

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 $\rightsquigarrow_{\mathsf{R}}^{*}$ environment interference on (an approximation t_1 of) C_1, and vice versa. Note that the pattern of rearranging subjective thread/environment components recurs at the level of triples $d = h; \mu; \sigma$: the parallel composition uses $(d_1 \oplus d_2, d_{\mathsf{O}})$ and the left and right child threads use $(d_1, d_2 \oplus d_{\mathsf{O}})$ and $(d_2, d_1 \oplus d_{\mathsf{O}})$, respectively.

Lemma 6 (Closure under parallel composition)**.** *If* $\mathsf{after}\ (d_1 \mid h_{\mathsf{R}} \mid d_2 \oplus d_{\mathsf{O}})\ t_1\ Q_1$ *and* $\mathsf{after}\ (d_2 \mid h_{\mathsf{R}} \mid d_1 \oplus d_{\mathsf{O}})\ t_2\ Q_2$, *then* $\mathsf{after}\ (d_1 \oplus d_2 \mid h_{\mathsf{R}} \mid d_{\mathsf{O}})\ (\mathsf{Par}\ t_1\ t_2\ \mathsf{Ret})\ (Q_1 \circledast Q_2)$, *where*

$$\begin{aligned} Q_1 \circledast Q_2 \ \widehat{=}\ \lambda c'\, v'.\, \exists d_1'\, d_2'\, h_{\mathsf{R}}'\, d_{\mathsf{O}}'.\ & c' = (d_1' \oplus d_2' \mid h_{\mathsf{R}}' \mid d_{\mathsf{O}}') \wedge \\ & Q_1\ (d_1' \mid h_{\mathsf{R}}' \mid d_2' \oplus d_{\mathsf{O}}')\ (v'.1) \wedge \\ & Q_2\ (d_2' \mid h_{\mathsf{R}}' \mid d_1' \oplus d_{\mathsf{O}}')\ (v'.2) \end{aligned}$$

Finally, the Frame lemma can be viewed as an instance of the Parallel lemma, by taking C_2 to be the idle thread that does not change the heap or auxiliaries.

Lemma 7 (Frame)**.** *If* $\mathsf{after}\ (d_1 \mid h_{\mathsf{R}} \mid d_2 \oplus d_{\mathsf{O}})\ t\ Q$, *then* $\mathsf{after}\ (d_1 \oplus d_2 \mid h_{\mathsf{R}} \mid d_{\mathsf{O}})\ t\ Q'$, *where*

$$\begin{aligned} Q' \ \widehat{=}\ \lambda c'\, v'.\, \exists d_1'\, h_{\mathsf{R}}'\, d_{\mathsf{O}}'.\ & c' = (d_1' \oplus d_2, h_{\mathsf{R}}', d_{\mathsf{O}}') \wedge \\ & Q\ (d_1' \mid h_{\mathsf{R}}' \mid d_2 \oplus d_{\mathsf{O}}')\ v \end{aligned}$$

Theorem 1 (Soundness)**.** *If* $\Gamma \vdash J$, *then* $[\![\Gamma \vdash J]\!]$.

Proof. By induction on the derivation of J. Each basic command is sound because the pre- and postconditions are stable under environment interference, the precondition implies the command is memsafe, and the resulting configuration satisfies the postcondition. The SEQ, PAR and FRAME rules are sound by Lemmas 5, 6, and 7. The fix rule is sound by the Knaster-Tarski theorem. The CONJ rule is sound by Lemma 3, and the CONSEQ rule by Lemma 4. 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. □

6. Related Work

Owicki and Gries' Resource Invariants (RI) [21] emphasizes a spatial specification of shared state that usually suffices for coarse-grained concurrency. Jones' Rely/Guarantee (RG) [12] emphasizes a temporal specification of thread and environment interference on shared state that is appropriate for fine-grained concurrency and can avoid some of the auxiliary state that appears in RI proofs.

The combination of RI or RG with Separation Logic in Concurrent Separation Logic (CSL) [19], RGSep [26], SAGL [9, 8], and Deny-Guarantee [7] has proved fruitful for the compositional verification of stateful, concurrent programs. However, to prove noninvariant properties as in the incrementor example, those logics still require auxiliary state to relate local program assertions to global (spatial or temporal) invariants. Auxiliary state makes the proof of a single thread sensitive to the global thread structure, which prevents local reasoning and leads to noncompositionality. In SCSL, subjective auxiliary state suffices to overcome this noncompositionality and recovers local reasoning in a CSL-style logic. On the other hand, RG specifications give fine-grained descriptions of possible interference, which to our knowledge isn't possible with subjective auxiliary state and RI alone.

Whereas CSL and existing variants use *local* specifications that only refer to the program's private state, SCSL gains expressivity through the a_{O} view to provide *non-local* information about the environment's auxiliary contribution, but retains CSL-style *local reasoning* by leveraging the PCM structure of auxiliaries to avoid sensitivity to the environment's thread structure.

Concurrent Abstract Predicates (CAP) [6] by Dinsdale-Young et al.'s is an axiomatic logic that *can* prove the incrementor and coarse-grained set *without* auxiliary state. A CAP proof involves defining a set of actions, which are RG-style transitions on private and shared state, which includes concrete heap and abstract capabilities that identify enabled actions (thus there is a subtle mutual recursion between an action and the capability to perform the action). Deductions in a proof use the actions to move heap and capabilities between the private and shared states. Intermediate assertions must be checked to be stable under the environment's action interference, which is standard in RG-style proofs. Whereas our prototype SCSL only provides one lock with global scope, CAP provides multiple shared regions each with an associated set of protocols (i.e., RG interference specifications).

Although CAP overcomes the need for auxiliary state, in our opinion the corresponding proofs of the incrementor and coarse-grained set are more indirect than the noncompositional RI proofs based on auxiliary state. Moreover, we consider the complexity of fine-grained interference reasoning should be unnecessary for coarse-grained concurrency, where critical sections are meant to abstract away from interference. To achieve compositional proofs of the same programs in SCSL, it suffices to pick a PCM and resource invariant, and to conduct RI-style sequential reasoning in the critical section. Although SCSL assertions go beyond private state in mentioning the environment's contribution a_{O}, it is only tracked in the critical section where it is known to remain fixed, so we can avoid reasoning about stability under interference.

Since SCSL follows the RI tradition, it is best suited for coarse-grained concurrency; by contrast, CAP is more general because the interference specifications yield compositional proofs for both coarse- and fine-grained concurrency. Moreover, abstract predicates enable **modularity** by giving different library implementations the same specification, thus hiding their internal concurrency. Since the SCSL embedding into CiC gives us access to abstraction over predicates (as well as over types and higher-order modules), SCSL already supports this kind of modularity. However, to reach the expressive power of CAP, we have to investigate how to combine subjectivity with RG for fine-grained concurrency.

Jacobs and Piessens' **fine-grained concurrency specifications** [11] provide an alternative means for modularity: a procedure can be verified parametrically in the caller's invariant, auxiliary state and code, which are then instantiated by each call site. This allows verifying the increment procedure once, but the two- and three-thread programs must use different invariants and each procedure call must use different auxiliary state and update code. In SCSL, we can also parametrize a procedure in the caller's invariant and auxiliaries as illustrated by the iterator. Moreover, subjectivity overcomes the need for different invariants and auxiliary state, and local auxiliary functions can update the auxiliary without sensitivity to the thread's identity.

To our knowledge, SCSL is the first axiomatic concurrency logic to support **higher-order procedures**, which is necessary to prove the iterator where the argument procedure exposes its effects and is general enough to support polymorphic instantiation (e.g., iter (iter xincr2) in Section 3.2). A first-order iterator has been proved in CAP [5], where the procedure applied to each element is hard-coded and doesn't expose any state or concurrency effects.

Fu et al.'s **HLRG** logic [10] combines rely/guarantee and temporal reasoning over explicit history traces. This eliminates the need for auxiliary state, but, as in CAP, at the expense of more involved reasoning than necessary in the case of coarse-grained algorithms. HLRG only handles a single top-level parallel composition of a fixed number of threads; by contrast, SCSL allows parallel composition in any command and the number of forked threads can vary dynamically, which is necessary to prove the iterator.

The denotational **semantics** of SCSL is inspired by Brookes' domain-theoretic model for CSL [3], but differs in several crucial respects. Brookes uses store (*a.k.a.* stack), heap, and resource (*a.k.a.* lock) actions to represent the operation and its result. We use actions to represent heap operations, and only by stepping an action do we obtain its result. We can avoid a stack altogether because our commands return values which are bound to immutable variables (*a.k.a.* identifiers). Since we implement the lock with a dedicated heap location, we avoid a separate notion of resource actions.

Brookes denotes a command as a set of *action traces*, which are *finite or infinite* sequences of actions including all possible results. For example, reading ℓ has the denotation $\{(\mathsf{Read}\ \ell, m)\}_{m\in\mathbb{Z}}$. We avoid enumerating all possible results by shifting to *action trees*: the Cons tree pairs an action (in our sense) with a continuation that takes the result of the operation and generates a tree for the rest of the computation. Since we are concerned with safety properties, it suffices to consider *finite* tree approximations.

Brookes' parallel composition is given by a *fairmerge* interleaving of traces. For example, reading ℓ in parallel has the denotation $\{(\mathsf{Read}\ \ell, m)\ (\mathsf{Read}\ \ell, n)\}_{m,n\in\mathbb{Z}}$, the case $m \neq n$ is a *non-sequential* trace that accounts for the environment mutating ℓ between the two reads and is needed for compositionality. We avoid dealing with non-sequential traces by including a Parallel tree constructor and unfolding the execution in the always predicate.

Brookes includes a fault state to account for races (i.e., one thread writes to a location while another thread also reads or writes), accessing a nonexistent location, and unlocking without restoring the resource invariant. We avoid an explicit fault state by ensuring that "well-specified programs don't go wrong": every heap operation encountered during an execution is safe to execute. We do not identify races because threads must interfere on the implicit lock location, but it is immediate from the interpretation of Hoare triples that parallel threads can otherwise only mutate their private heaps, which are disjoint.

Besides Brookes' denotational model, Vafeiadis [25] and Feng et al. [9] give operational proofs of CSL's soundness. We are similar to those approaches in two ways: (1) the explicit use of RG-style specifications in the semantics to describe interference over the shared resource, and (2) the use of step-indexing in the definition of safety, although we use tree paths whereas they use natural numbers. Since we interpret SCSL Hoare triples as monadic functions in CiC, however, operational methods do not suffice and we must resort to a semantic proof of soundness.

Existing approaches to the soundness of CSL use a *small* footprint in the sense that their semantic definitions of safety only refer to the thread's private state and (in some cases) the shared state. Similarly, the combination of RG and Separation Logic in RGSep [26] and SAGL [9, 8] only describe interference over the shared state. By contrast, our subjective rely/guarantee transitions use a *large* footprint, which encompasses the shared state as well as the thread and environment's private state of heap, lock ownership, and user-chosen auxiliary state. While our subjective rely/guarantee transitions are hard-coded for RI, we expect that the logic can be generalized to user-chosen RG interference and thus we will be able to verify fine-grained concurrency.

The concept of **local action** appears in Abstract Separation Logic [4] to model stateful, non-deterministic programs. A local action is a function $f : \Sigma \to \mathcal{P}(\Sigma)^\top$ where Σ is a *separation algebra* (i.e., a *cancellative* PCM) of states σ. The codomain is a set of possible next-states to express nondeterminism or an error marker $\top$ to indicate that the action faults. The function must satisfy the *locality condition*: if $\mathsf{valid}\ (\sigma_1 \oplus \sigma_2)$, then $f(\sigma_1 \oplus \sigma_2) \sqsubseteq \{\sigma_1' \oplus \sigma_2 \mid \sigma_1' \in f(\sigma_1)\}$. By contrast, our local auxiliary functions Φ are deterministic because there is at most one possible next-state and we implicitly avoid the error marker by using the validity conditions. Since auxiliary *functions* can be noncomputable, we conjecture they will enable more expressive specifications than the traditional use of (computable) auxiliary *code*.

Krishnaswami et al. [15] use a PCM to abstractly represent the states of various program modules in a setting of a sequential higher-order language. Similar to our resource invariant, their module invariants are also predicates over the combined abstract states of all the modules.

7. Conclusion and Future Work

We propose Subjective Concurrent Separation Logic (SCSL) as a combination of Resource Invariants and Separation Logic with *subjective auxiliary state*, which tames the objective thread structure exposed by classical auxiliary state. The approach is enabled by generalizing auxiliaries from stack and heap to user-chosen partial commutative monoids (PCM), and from auxiliary code to local auxiliary functions. Each thread has a subjective perspective that splits auxiliary state into contributions by the *self* (the thread itself) and the *other* (its environment). The auxiliary PCM crucially permits agglomerating all environment thread's contributions, which makes proofs insensitive to the internal concurrency of the environment. SCSL provides parallel composition and frame rules that use the *subjective separating conjunction* to split auxiliary contributions in a coherent manner; surprisingly, the rules are sound even if the auxiliary PCM is not cancellative.

For an incrementor and a coarse-grained set with logically disjoint interference, we show that suitable choice of auxiliary PCM yields simple, compositional proofs independent of the number of parallel threads; by contrast, existing logics either use auxiliary state which makes proofs sensitive to the global thread structure, or achieve compositionality at the expense of complex Rely/Guarantee reasoning. We also prove a higher-order iterator with dynamic thread creation, whose verification is independent of the internal concurrency of the function applied to each element. In future work, we will study how subjective auxiliary state scales to more complex, real-world programs.

We prove the soundness of SCSL by a shallow embedding into the Calculus of Inductive Constructions using a novel denotational semantics of action trees and a definition of safety that uses rely/guarantee transitions over a large subjective footprint encompassing heaps, lock ownership, and user-chosen auxiliaries.

Since SCSL provides a single lock with global scope, a proof is committed to a single auxiliary PCM and resource invariant. In future work, we will scale the logic to multiple locks with delimited scope and refinement across auxiliary PCMs and resource invariants. We will also study the combination of subjective auxiliary state Rely/Guarantee reasoning for fine-grained programs in the presence of richer language constructs such as separate fork and join commands.

Acknowledgments

This research has been partially supported by Spanish MICINN Project TIN2010-20639 Paran10; AMAROUT grant PCOFUND-GA-2008-229599; and Ramon y Cajal grant RYC-2010-0743.

References

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

[2] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in separation logic. *ENTCS*, 155, 2006.

[3] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3), 2007.

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

[5] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, and Mark J. Wheelhouse. A simple abstraction for complex concurrent indexes. In *OOPSLA*, 2011.

[6] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[7] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.

[8] Xinyu Feng. Local rely-guarantee reasoning. In *POPL*, 2009.

[9] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.

[10] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, 2010.

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

[12] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.

[13] Cliff B. Jones. The role of auxiliary variables in the formal development of concurrent programs. Technical Report CS-TR-1179, University of Newcastle upon Tyne, Computing Science, 2009.

[14] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11, 1999.

[15] Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially substructural types. In *ICFP*, 2012.

[16] Ruy Ley-Wild and Aleksandar Nanevski. Supporting Material. https://software.imdea.org/˜aleks/scsl/, July 2012.

[17] Per Martin-Löf. Haupstatz for the intuitionistic theory of iterated inductive definitions. In *Scandinavian Logic Symposium*, 1971.

[18] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[19] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3), 2007.

[20] Susan S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, 1975.

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

[22] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *POPL*, 2005.

[23] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare logics. In *LICS*, 2006.

[24] Uday S. Reddy and John C. Reynolds. Syntactic control of interference for separation logic. In *POPL*, 2012.

[25] Viktor Vafeiadis. Concurrent separation logic and operational semantics. *ENTCS*, 276, 2011.

[26] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.