

Subjective Concurrent Separation Logic

Ruy Ley-Wild and 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 program logics for concurrency have a limited form of compositionality. Proving non-trivial properties usually requires the use of auxiliary state, which is “objective” in the sense that each thread’s auxiliary state is given a globally-unique name. Since auxiliary state exposes the program’s global structure to each local thread, axiomatic approaches fail to be compositional in the presence of auxiliary state.

We propose “subjective” auxiliary state as a solution to this historical limitation of axiomatic program logics. Each thread can be verified with a subjective view on auxiliary state: auxiliary state can be partitioned to either belong to the *self* (i.e., the thread itself) or to the *other* (i.e., its environment). We formulate Subjective Concurrent Separation Logic as a combination of the resource invariant method, separation logic, and subjective auxiliary state for a first-order, stateful, concurrent language. The logic provides compositional verification even when auxiliary state is needed. We give an operational proof of logic’s soundness.

I. INTRODUCTION

Auxiliary state (*a.k.a.* ghost state) is a necessary evil that compensates for the “expressive weakness” [1] of axiomatic program logics for concurrency. It is necessary to partially expose the internal behavior of threads in order to relate program invariants to pre- and postconditions. But it is evil because it breaks the compositionality of all of the logics that use it. This weakness applies to early axiomatic methods such as Owicki and Gries’ interference freedom [2] and resource invariants [3], and Jones’ rely/guarantee [4]. The modern descendants Concurrent Separation Logic (CSL) [5], RGSep [6], SAGL [7], concurrent abstract predicates [8] and fine-grained concurrency specifications [9] use separation logic to aid the treatment of state and enable verification that respects module boundaries, but inherit their ancestors’ expressive weakness.

We illustrate the problem using the canonical example due to Owicki and Gries [3] in the setting of CSL. The concurrency model enforces a *mutual exclusion protocol* by confining access to the *shared state* to a *critical section*: the program must first lock to acquire the shared state satisfying a *resource invariant* I , and then it can mutate the shared state and its own auxiliary. Generally, the mutation in the critical section may violate I , but I must be restored before unlocking.

We want to prove that two threads that concurrently increment a shared pointer x have the overall effect of incrementing its value by 2. An invariant for the shared state that *only* refers to x has no way to identify each thread’s contribution to the total value, thus, at best it can show that x remains non-negative. Owicki and Gries’ solution (and other methods follow suit) is for each thread to have *auxiliary state* (pointer

y or z) that exposes its individual *contribution* to x ; auxiliaries are updated for the sake of verification, but aren’t needed for execution. Using notation based on CSL [5] with fractional permissions [10], the *binary* invariant $I_2(a, b) \triangleq x \mapsto a + b * y \mapsto a * z \mapsto b$ describes that the shared state x contains the sum $a + b$ and the auxiliaries have corresponding contributions.¹ Ownership of an auxiliary is split in half between the invariant and the owning thread. The proof outline instruments the program code with **shaded** code for updating auxiliaries, and with **braced** assertions that describe what holds at specific program points. The final assertion and the invariant imply that x increments by 2.

$$\begin{array}{c}
 \{y \mapsto a\} \\
 \text{lock;} \\
 \{y \mapsto a * \exists \beta. I_2(a, \beta)\} \\
 x := [x] + 1; \text{ } y := [y] + 1; \\
 \{y \mapsto a + 1 * \exists \beta. I_2(a + 1, \beta)\} \\
 \text{unlock} \\
 \{y \mapsto a + 1\}
 \end{array}
 \parallel
 \begin{array}{c}
 \{y \mapsto a * z \mapsto b\} \\
 \{z \mapsto b\} \\
 \text{lock;} \\
 \{z \mapsto b * \exists \alpha. I_2(\alpha, b)\} \\
 x := [x] + 1; \text{ } z := [z] + 1; \\
 \{z \mapsto b + 1 * \exists \alpha. I_2(\alpha, b + 1)\} \\
 \text{unlock} \\
 \{z \mapsto b + 1\}
 \end{array}
 \quad \parallel \quad
 \begin{array}{c}
 \{y \mapsto a + 1 * z \mapsto b + 1\}
 \end{array}$$

If instead we wanted to prove the effect of three parallel increments, we would have to use a different *ternary* invariant $I_3(a, b, c) \triangleq x \mapsto a + b + c * y \mapsto a * z \mapsto b * w \mapsto c$. 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 CSL, incrementation lacks a single proof that can be reused in different contexts.

In this sense, axiomatic proofs with auxiliaries fail to be *compositional* because they lack a *substitution principle*: it is not possible to prove a subprogram and reuse its proof directly in a larger context. The problem is that since auxiliaries are globally-unique pointers, the invariant has an *objective* view of auxiliary state which exposes the global thread structure to the verification of each local thread. Thus verification is specific to the number and the pattern in which threads are forked. Even worse, when threads are forked dynamically (e.g., a program iterates over a list and forks a separate thread to process each element), auxiliary state is not determined statically and must be represented as a linked structure in the heap simply in order to define I . The structure has to reflect the order of forking, so that threads can address the mirroring subheaps, all of which makes the invariant very context-specific. The

¹The separating conjunction $p * q$ means that the heap splits into disjoint subheaps that satisfy p and q that are passed to each subthread of a parallel composition; $g \mapsto v$ is a *half fractional permission* that allows dereferencing g ; $g \mapsto v * g \mapsto v$ is equivalent to $g \mapsto v$ and allows updating g .

problem is further exacerbated if the iterated thread itself forks new threads, and thus, presumably, has to dynamically allocate its own auxiliaries.

To achieve compositionality in the presence of auxiliary state, the first step is therefore to make the resource invariants of programs *insensitive* to the number of threads and the order in which they are forked. To that end, in this paper we propose a *subjective* perspective on auxiliary state. While there exists a single, global, auxiliary state, say α , which holds the combined contributions of *all* threads, each thread has two complementary *views* of it: a *self* view a_S of the thread's auxiliary contribution, and an *other* view a_O of the combined contribution of all the other threads. The views are local to a proof of each thread, but in each of them, $a_S + a_O = \alpha$. For example, in the proof of two increments, the initial view of the contributions (a_S, a_O) of the left subproof is (a, b) , while for the right subproof, it is (b, a) . In a proof of three increments, the views for the three threads would be $(a, b + c)$, $(b, a + c)$, and $(c, a + b)$.

With the subjective view, we can have a *unary* resource invariant $I_*(\alpha)$, independently of the number of parallel threads. In the increment example, we can choose $I_*(\alpha) = x \mapsto \alpha$. Mutual exclusion enforces that $I_*(a_S + a_O)$ holds after locking and before unlocking, but in each thread's proof, a_S and a_O are interpreted *according to that thread's subjective view*.

While subjectivity makes the invariant insensitive to the global thread structure, it doesn't immediately lead to compositionality. The critical point concerns how separate proofs of two threads can be combined into a proof of the parallel composition. If one thread updates its contribution a_S , how do we ensure that the proof of the other thread remains a valid subproof of the combination, when it sees its a_O change?

In this paper, we resolve this question by giving a sound rule of parallel composition that 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 infer properties of parallel composition, while treating the proofs of component threads as black boxes. In particular, we make the following contributions.

(1) We observe that the auxiliary state can be cast algebraically as a *partial commutative monoid* (PCM); the algebraic description shows how to split and recombine auxiliary views during forking and joining (Section III). PCMs have been used previously to model heaps, but we recognize that the *values* stored in auxiliary variables also form a PCM.

(2) We formulate Subjective Concurrent Separation Logic (SCSL) as a combination of the resource invariants method, separation logic, and subjectivity for a first-order, stateful, concurrent language with one lock (Section IV).

(3) We give SCSL proofs of incrementation examples, including the iterator; to the best of our knowledge, these are the first compositional proofs for such programs (Section II).

(4) We prove SCSL operationally sound and compositional, i.e., it satisfies the substitution principle on threads and their proofs (Section V).

II. OVERVIEW BY EXAMPLE

In SCSL, assertions are predicates over *worlds* w , which are quadruples of the form $w = h_S; m_S; a_S; a_O$. Here h_S is the thread's private heap (finite maps from pointers to values), m_S is the thread's subjective view of lock ownership (either Own if it owns the lock, or NOwn), and a_S and a_O are the thread's view of self and other's auxiliary contributions. The most common SCSL assertion is $[m, a, b] p$, which holds in w if $m_S = m$, $a_S = a$, $a_O = b$, and h_S satisfies the separation logic heap predicate p .

A command C is a (possibly) stateful and concurrent program that 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 .

We use a *function triple*, $\forall x:A. \{p\}F(x) : B\{q\}$, to specify a (potentially recursive) first-order command function F of the form $\text{fix } f.x.C$ from type A to type B , where p and q may depend on x . We admit basic types such as unit, nat, bool, and pointers ptr (isomorphic to nat), all with the usual values. We provide cartesian product types $A_1 \times A_2$, to allow for functions with more than one argument or result.

A. Incrementation

The incrementation program $\text{incr}(\iota)$ can be specified in SCSL, with the *unary* resource invariant $I_*(\sigma) = x \mapsto \sigma$.

$\forall \iota:\text{nat}. \{[\text{NOwn}, \alpha, -] \text{emp}\} \text{incr}(\iota) : \text{unit}\{[\text{NOwn}, \alpha + \iota, -] \text{emp}\}$

incr is safe to run with an empty private heap (thus, it's safe in any larger heap), and the private heap is unchanged upon termination, but the thread's contribution increments by ι ; α is a *logical variable* that scopes over the pre- and postcondition to relate the initial and final states; the notation $-$ in the assertion abbreviates $\exists \beta. a_O = \beta$ to indicate that the other's contribution remains unknown. The following code and proof outline for incr are parametric in ι and α . Note that unlock is annotated by a mathematical function $\Phi(a_S) = a_S + \iota$ to update the auxiliary contribution a_S .

	$\{[\text{NOwn}, \alpha, -] \text{emp}\}$	precondition
1	lock;	
	$\{\exists \beta. [\text{Own}, \alpha, \beta] I_*(\alpha + \beta)\}$	by lock
	$(\exists \beta. \{[\text{Own}, \alpha, \beta] x \mapsto (\alpha + \beta)\})$	by EXIST
2	$v \leftarrow [x];$	
	$\{[\text{Own}, \alpha, \beta] x \mapsto (\alpha + \beta) \wedge v = \alpha + \beta\}$	by read
3	$x := v + \iota;$	
	$\{[\text{Own}, \alpha, \beta] x \mapsto ((\alpha + \iota) + \beta)\}$	by write
	$\{[\text{Own}, \alpha, \beta] I_*(\alpha + \iota + \beta)\}$	by CONSEQ
4	unlock $_{\Phi}$	
	$\{[\text{NOwn}, \alpha + \iota, -] \text{emp}\}$	by unlock
	$\{\exists \beta. [\text{NOwn}, \alpha + \iota, -] \text{emp}\}$	by EXIST
	$\{[\text{NOwn}, \alpha + \iota, -] \text{emp}\}$	by CONSEQ

The lock operation (line 1) changes the lock status from NOwn to Own and uses an existential quantifier to name the environment's contribution a_O for the duration of the critical section. By mutual exclusion, this contribution remains unchanged, while incr is in a critical section, because the environment can't access the shared state and change its own contribution.

From before line 2 to after line 4, we use the Hoare proof rule for existentials to introduce β as a name for the other's contribution, to expose that a_0 remains fixed throughout the critical section. The program gains ownership of the shared state satisfying the invariant $I_*(\alpha + \beta)$ where α stands for the thread's initial contribution. Next, the dereference (line 2) binds the contents of x (which equals $\alpha + \beta$ by the invariant) to the local variable v , and extends the assertion with an equation that describes v . Next, the update (line 3) increments x by ι , which is reflected in the assertion about the contents of x . Dereference as well as update preserve the lock status and both auxiliary contributions. Therefore, immediately before unlock (line 4), the shared state modified in the critical section satisfies $I_*(\alpha + \iota + \beta)$. unlock changes a_5 from α to $\alpha + \iota$, as determined by the function Φ , after which $I_*(a_5 + a_0)$ again holds. Thus, the invariant on the shared state is restored, and the pointer x may be returned to the resource (i.e., incr 's heap becomes empty). Finally, the existential β is again explicitly quantified inside the assertion and since it doesn't appear in the assertion, we can consequently eliminate the quantifier.

The subjective perspective in SCSL provides single handles a_5 and a_0 for the thread's and environment's contributions, which gives the proof of incr two advantageous properties. First, the update function Φ applies to a_5 without knowledge of the thread's identity, whereas CSL proofs update specific auxiliaries in each thread (e.g., y in the left thread, and z in the right thread in Section I). Second, the two views enable a unary invariant that avoids sensitivity to the internal structure of the environment, whereas CSL proofs expose the environment threads in the invariant (e.g., $z \vdash \beta$ in the left thread and $y \vdash \alpha$ in the right thread in Section I). Thus, multiple threads can instantiate the proof, each with its own subjective perspective (a_5, a_0), without having to change the auxiliary code, invariant, or proof.

Moreover, notice that m_5 , a_5 and a_0 are not program variables and don't appear in the auxiliary code Φ , thus there is no way for the program to directly change them. The update function Φ upon unlock can indirectly change a_5 , but not m_5 or a_0 . Permissions [11], [12], [13] can be used to control which auxiliaries can be changed by auxiliary code, but in SCSL this discipline arises merely as a consequence of subjectivity. In Section V, we show that the auxiliary annotations don't influence the operational semantics of the program.

The SCSL proof of incr is compositional: we can instantiate α and ι to show that $\text{incr}(i) \parallel \text{incr}(j)$ increments x by $i+j$.

$$\begin{array}{c} \{[\text{NOwn}, a+b, -] \text{emp}\} \\ \{[\text{NOwn}, a, -] \text{emp} \otimes [\text{NOwn}, b, -] \text{emp}\} \\ \{[\text{NOwn}, a, -] \text{emp}\} \parallel \{[\text{NOwn}, b, -] \text{emp}\} \\ \text{incr}(i) \parallel \text{incr}(j) \\ \{[\text{NOwn}, a+i, -] \text{emp}\} \parallel \{[\text{NOwn}, b+j, -] \text{emp}\} \\ \{[\text{NOwn}, a+i, -] \text{emp} \otimes [\text{NOwn}, b+j, -] \text{emp}\} \\ \{[\text{NOwn}, (a+i)+(b+j), -] \text{emp}\} \end{array}$$

The parallel composition starts with a self contribution $a_5 = a+b$. Whereas CSL parallel composition uses the separating conjunction $*$ to split the *heap* among parallel threads, SCSL's *subjective separating conjunction* $\otimes$ splits both the

heaps and the auxiliary *views* in a *coherent* manner (cf. Section IV). When a parent thread forks two children threads, the parent's self view a_5 splits between its children, while the children's a_0 's are induced to preserve coherence (i.e., the left child's environment includes the right child, and vice-versa). Thus, the combination $a_5 + a_0$ of each viewer's view is the same irrespective of the viewer.

viewer:	parent $\Leftrightarrow$ child ₁ $\otimes$ child ₂		
view:	self a_5	$a+b$	a
	other a_0	c	$b+c$
combined view:	$a_5 + a_0$	$a+b+c$	

In the above proof, a_0 starts as unknown, and thus remains so in both children threads. The left thread instantiates the incr proof with a and i to show that the overall contribution increments from a to $a+i$; and analogously for the right thread. Crucially, both children threads instantiate the same proof of incr without changing the auxiliary code or invariant. The soundness of parallel composition ensures that, dually, at the join point the children's subelves are combined back into the parent's self: the subthreads' self's contributions are added and the parallel composition ends with a_5 incremented by $i+j$ over the initial value.

We can directly reuse the SCSL proof of the two-thread program compositionally in a three-thread program without having to change the invariant, auxiliary update, or subproof for the particular context. The three-thread program increments the initial contribution $a+b+c$ by $i+j+k$ with a structurally similar proof as the two-thread program.

$$\begin{array}{c} \{[\text{NOwn}, a+b+c, -] \text{emp}\} \\ \{[\text{NOwn}, a+b, -] \text{emp} \otimes [\text{NOwn}, c, -] \text{emp}\} \\ \{[\text{NOwn}, a+b, -] \text{emp}\} \parallel \{[\text{NOwn}, c, -] \text{emp}\} \\ \text{incr}(i) \parallel \text{incr}(j) \parallel \text{incr}(k) \\ \{[\text{NOwn}, (a+i)+(b+j), -] \text{emp}\} \parallel \{[\text{NOwn}, c+k, -] \text{emp}\} \\ \{[\text{NOwn}, (a+i)+(b+j), -] \text{emp} \otimes [\text{NOwn}, c+k, -] \text{emp}\} \\ \{[\text{NOwn}, (a+i)+(b+j)+(c+k), -] \text{emp}\} \end{array}$$

Moreover, we can substitute programs with proofs of the same specification. For example, we can replace the parallel $\text{incr}(i) \parallel \text{incr}(j)$ by the sequential $\text{incr}(i); \text{incr}(j)$ (different number of threads) or $\text{incr}(i+j)$ (different number of critical sections) and the proof remains the same.²

B. A List Iterator

In SCSL, we can prove an iterator iter that applies a function f to each element of a list ns .³ The specification context indicates that f that takes an argument n and has the effect of incrementing the initial local auxiliary contribution 0 by $\phi(n)$:

$$\begin{array}{l} \forall n:\text{nat}. \{[\text{NOwn}, 0, -] \text{emp}\} f(n):\text{unit} \{[\text{NOwn}, \phi(n), -] \text{emp}\} \\ \vdash \forall ns:\text{list nat}. \{[\text{NOwn}, 0, -] \text{emp}\} \text{iter}(ns):\text{unit} \{[\text{NOwn}, \phi(ns), -] \text{emp}\} \end{array}$$

²Strictly speaking, the sequential composition returns unit whereas the parallel version returns $\text{unit} \times \text{unit}$, so the latter must be coerced into unit to have the same type by sequentially composing with $\text{ret}()$.

³We assume the language includes pure lists and pattern-matching.

The proof outline of iter:

```

fix iter. ns.
match ns with
| Nil => {[NOwn, 0, -] emp} ret () {[NOwn,  $\bar{\phi}(\text{Nil})$ , -] emp}
| Cons n' ns' =>
  (
    (
      {[NOwn, 0, -] emp}
      f n'
      {[NOwn,  $\phi(n')$ , -] emp}
      {[NOwn,  $\phi(n') + \bar{\phi}(ns')$ , -] emp}
      {[NOwn,  $\bar{\phi}(\text{Cons } n' ns')$ , -] emp}
      ret ()
      {[NOwn,  $\bar{\phi}(ns)$ , -] emp}
    )
    ||
    (
      {[NOwn, 0, -] emp}
      iter ns'
      {[NOwn,  $\bar{\phi}(ns')$ , -] emp}
    )
  )

```

The overall effect of iter is to increment the local auxiliary contribution by $\sum_{n \in ns} \phi(n)$, which we abbreviate as $\bar{\phi}(ns)$. iter is defined *recursively* by fix with two arguments: iter for recursive calls, and the list ns . The function body pattern matches the list: if it's Nil, then it returns unit and the overall effect increments the local contribution by $\bar{\phi}(\text{Nil}) = 0$; otherwise if it's Cons n' ns' , it concurrently applies f to the head element n' and recurses on the tail ns' , then returns unit, so the contribution increments by $\phi(n') + \bar{\phi}(ns') = \bar{\phi}(\text{Cons } n' ns')$.

Although iter has a dynamic forking pattern that depends on its list argument ns and although f may spawn multiple threads, the subjective view on auxiliary contributions enables verifying the iterator *uniformly* for any functions f that matches the stipulated specification and any input list ns . By SCSL's substitution principle, we can deduce that both $[\text{fun } n. \text{incr}(n); \text{incr}(n)/f] \text{iter}$ and $[\text{fun } n. \text{incr}(n) \parallel \text{incr}(n)/f] \text{iter}$ have iter's specification with $\phi(n) = n + n$, but without the hypothetical assumption f . Here $\text{fun } n.C$ abbreviates a non-recursive function $\text{fix } f.n.C$.

III. PARTIAL COMMUTATIVE MONOIDS

In the case of incr (Section II), the auxiliaries a_s and a_o were natural numbers. In general, however, each thread and its environment may have a different number of auxiliaries of different types. We support the general case of auxiliary state by shifting from scalars to structures such as tuples, tagged unions, and inductive types. The structure must satisfy the algebraic properties of a PCM. A PCM is a triple $(U, \bullet, \text{id})$, consisting of a set U , a *partial* binary operation $\bullet$ on U , that is commutative and associative, and has a unit element id .

Why are PCMs the required algebraic structure? The unit id captures the contribution of a thread that does not change the shared resource. The operation $\bullet$ expresses the joint contribution of two parallel threads, thus, it must be commutative and associative to mirror parallel composition itself. Partiality is necessary to indicate that the combination is undefined on incompatible contributions (illustrated below).

We have implicitly used three PCMs so far in this paper. The first is the PCM of natural numbers with addition $(\text{nat}, +, 0)$. The second is the PCM of heaps: the unit is the empty heap and the binary operation is disjoint union $h_1 \uplus h_2$, which is undefined if the heaps overlap. The third is the PCM for lock ownership, whose values populate the m_s components of the worlds. The underlying set owns has two elements, Own and

$h_s; m_s; a_s; a_o \models \text{emp}$	iff $h_s = \text{empty}$
$h_s; m_s; a_s; a_o \models p_1 * p_2$	iff $h_1; m_s; a_s; a_o \models p_1$ and $h_2; m_s; a_s; a_o \models p_2$ for some h_1 and h_2 s.t. $h_s = h_1 \bullet h_2$
$h_s; m_s; a_s; a_o \models x \mapsto v$	iff $h_s = x \leadsto v$
$h_s; m_s; a_s; a_o \models [m, a_1, a_2] p$	iff $(m_s, a_s, a_o) = (m, a_1, a_2)$ and $h_s; m; a_1; a_2 \models p$
$h_s; m_s; a_s; a_o \models p_1 \otimes p_2$	iff $h_1; m_1; a_1; a_2 \bullet a_o \models p_1$ and $h_2; m_2; a_2; a_1 \bullet a_o \models p_2$ for some $h_1, h_2, m_1, m_2, a_1, a_2$ s.t. $h_s = h_1 \bullet h_2, m_s = m_1 \bullet m_2, a_s = a_1 \bullet a_2$

Fig. 1. Semantics of assertions $h_s; m_s; a_s; a_o \models p$ (fragment).

NOwn, and the binary operation is defined as

$$x \bullet y = \begin{cases} \text{NOwn} & \text{if } (x, y) = (\text{NOwn}, \text{NOwn}) \\ \text{Own} & \text{if } (x, y) = (\text{Own}, \text{NOwn}) \text{ or } (\text{NOwn}, \text{Own}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The definition captures the intuition that a parallel composition of two threads owns a lock if exactly one of the threads owns it, thus the unit is NOwn. Due to mutual exclusion, $\text{Own} \bullet \text{Own}$ is undefined because both threads cannot own the lock. With the unified view of heaps, lock ownership and auxiliary state as PCMs, we can now present the formal semantics of SCSL assertions in Figure 1.

We will use three standard constructions over PCMs. First, given two PCMs, their cartesian product is also a PCM, with $\bullet$ and id defined pointwise. Second, an arbitrary set A can be lifted into a PCM option A , with values None or Some v where v has type A ; the binary operation satisfies $\text{None} \bullet x = x = x \bullet \text{None}$ ($\text{Some } u \bullet \text{Some } v$ is undefined) and the unit is None. Third, given a PCM $(U, \bullet, \text{id})$, we define $U^\top = (U \cup \{\top\}, \bullet_\top, \text{id})$, where $\bullet_\top$ is the following *total* function:

$$x \bullet_\top y = \begin{cases} x \bullet y & \text{if } x, y \neq \top \\ \top & \text{otherwise} \end{cases}$$

We think of $\top$ as the *undefined* value. By convention, from now on, all our functions on PCMs, including $\bullet$, will be total; we explicitly employ $U^\top$ and have functions return $\top$ in case of undefinedness. Given $x \in U$, we write *valid* x iff $x \neq \top$; if $U \neq V^\top$, then every $x \in U$ is trivially valid.

We next argue that every auxiliary state (in the customary sense) can be cast as a PCM. Consider the following construction. Without loss of generality, we can assume each thread t_i uses a single auxiliary of some type A_i ; if it uses multiple auxiliaries we can create a tuple of their values. Next, we can construct the PCM (option $A_1 \times \dots \times \text{option } A_n$) for threads $1..n$; when the number of threads varies dynamically, we can use a tree of option A_i 's to mirror the forking structure, if necessary. Thus, if thread t_i 's auxiliary contribution is v_i , then its PCM representation is the tuple with $(\dots, \text{None}, \text{Some } v_i, \text{None}, \dots)$ with all Nones, except for Some v_i in the i th entry. If distinct threads i and j contribute v_i and v_j , then PCM representation is the tuple with Some v_i in the i th and Some v_j in the j th, and Nones elsewhere. However, if two threads attempt to contribute to the same i th entry, then the combination will be undefined in the

i th entry (by the option PCM definition) and thus the tuple combination will also be undefined (by the Cartesian product PCM definition). This construction reflects the forking structure, which is sometimes natural for the problem. Oftentimes, however, more optimized PCM's are possible, which avoid distinguishing each thread's contribution. This is the case in the increment example, where the PCM of natural numbers only keeps the total contribution of all threads because their identity doesn't matter.

We close the section with the definition of an important class of *local functions* on PCMs.

Definition 1. A total function Φ on a PCM U is local, written $\Phi:U \rightarrow_L U$, if for all $x, y \in U$, $\text{valid}(x \bullet y)$ and $\text{valid}(\Phi(x))$ imply $\Phi(x \bullet y) = \Phi(x) \bullet y$ and $\text{valid}(\Phi(x) \bullet y)$.

A local function Φ is insensitive as to how its argument may be split into the form $x \bullet y$; Φ is supposed to act on x only, but simply propagate y without changes. This property makes local functions appropriate for modelling the auxiliary code (e.g., as in Section II), that describes how individual threads modify their auxiliary state.

The intuition behind Definition 1 is as follows. Consider a thread t which forks into t_x and an immediately-terminating thread $\text{ret}()$. The forking splits the initial auxiliary state of t into $x \bullet y$, assigning the x portion to t_x , and the y portion to $\text{ret}()$. Now suppose that Φ models how t_x modifies its auxiliary state; that is, the ending auxiliary state of t_x is $\Phi(x)$. Then the ending auxiliary state of t will be $\Phi(x) \bullet y$. However, since t_x and t are equivalent (differing only in parallel composition with an idle thread), Φ models how t modifies its auxiliary state as well. Thus, the ending auxiliary state of t must also equal $\Phi(x \bullet y)$, leading to the equation from Definition 1.

In other words, a local function models only the changes to auxiliary state of the local thread, and is unaffected by, and doesn't affect, the auxiliary state of other threads running in parallel. Similarly, the side conditions about valid in Definition 1 model that not only the value, but also the validity of $\Phi(x)$ remains unaffected by joining with y .

IV. LOGIC

In the tradition of axiomatic program logics, the language of SCSL splits into the purely functional, divergence-free, fragment of *expressions* e (v when the expression is a value), and the imperative fragment of *commands* C . Expressions are classified by types A , while commands are specified by means of Hoare triples.

Like in a functional language, but unlike in a simple imperative language, our commands return values. This enables avoiding mutable variables (i.e., a stack) in favor of immutable variables, which one can substitute with expressions. A stack imposes a number of technical difficulties in a Hoare-style logic [11], [12], [13], so we prefer to avoid it, especially as it's not essential for our considerations here.

Thus, our syntax very much resembles monadic languages such as Haskell. A command can be a memory operation, a

lock or unlock operation, a monadic unit $\text{ret } v$ that returns v and terminates, monadic bind (i.e., sequential composition) $x \leftarrow C_1; C_2$, that runs C_1 then substitutes its result v_1 for x to run C_2 (we write $C_1; C_2$ when $x \notin \text{FV}(C_2)$), fork-join parallel composition $C_1 \parallel C_2$, or a conditional. We also include a fixed-point combinator for defining functions, and a function application $F(e)$.

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

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

Γ does not store the *logical variables*, which are left implicit in the pre- and postconditions. Thus, we assume that each occurrence of a logical variable comes annotated with a type (as in $x^{\text{nat}} + y^{\text{nat}}$), but will suppress such types in most cases, when they can be inferred from the context.

All SCSL judgments are parametric in the choice of the PCM of auxiliary state U , and the resource invariant I_* , subject to several semantic constraints on I_* inherited from CSL, and described in Section V. For the sake of simplicity, in this paper we consider only one dedicated lock, and thus only one PCM and resource invariant.⁴

Figure 2 presents the inference rules of SCSL. They are grouped into four judgments. The judgments for Hoare triples $\Gamma \vdash \{p\} C : A \{q\}$, and $\Gamma \vdash \forall x:A. \{p\} F(x) : A \{q\}$ have already been illustrated in Section II. The typing judgment for expressions $\Gamma \vdash e : A$ is standard from simply-typed lambda calculus, so we omit its rules. The judgment $\Gamma \vdash (p_1, q_1) \sqsubseteq (p_2, q_2)$ that serves as a side condition on the rule of consequence is intimately related to our treatment of logical variables, so we postpone its description until Section V. In the judgments, the postcondition q is an assertion potentially containing a dedicated variable res which binds the return result of the command.

We proceed to describe selected inference rules of SCSL.

a) *Memory commands*: We only describe the inference rule for read , as the other memory commands are similar. read takes a pointer x to dereference, and requires by its precondition a heap in which x is allocated, and points to some value v of type A . read is parametric in the lock status, and local and environment contributions, so its precondition names them with the unconstrained logical variables m , a and b . In the postcondition, the heap remains unchanged ($x \mapsto v$), the return result res is v . The lock status m and local contribution a to the shared resource are unchanged. However, the environment's contribution is subject to more complicated specification. It either remains fixed if read is invoked in a critical section, i.e., when $m = \text{Own}$, or may change in an unspecified way, if read is invoked outside the critical section.

⁴We defer to future work incorporating other features such as multiple locks or dynamic lock creation that are orthogonal to auxiliary state. These add bulk to the development, but conceptually merely require providing more complicated constructions over PCMs (e.g., have collections of PCMs that can be extended dynamically).

$\Gamma \vdash \forall x:A.$ $\Gamma \vdash \forall x:\text{ptr.}$ $\Gamma \vdash \forall x:\text{ptr.}$ $\Gamma \vdash \forall x:\text{ptr.}\forall y:A.$ $\Gamma \vdash$ $\Gamma \vdash$ $\Gamma \vdash \forall x:A.$	$\{[m, a, b] \text{ emp}\}$ $\{[m, a, b] x \mapsto -\}$ $\{[m, a, b] x \mapsto v\}$ $\{[m, a, b] x \mapsto -\}$ $\{[\text{NOwn}, a, -] \text{ emp}\}$ $\{[\text{Own}, a, b] I_* (\Phi(a) \bullet b)\}$ $\{[m, a, b] \text{ emp}\}$	$\text{alloc } x : \text{ptr}$ $\text{dealloc } x : \text{unit}$ $\text{read } x : A$ $\text{write } x \ y : \text{unit}$ $\text{lock} : \text{unit}$ $\text{unlock}_\Phi : \text{unit}$ $\text{ret } x : A$	$\{[m, a, \text{if } m = \text{Own then } b \text{ else } -] \text{ res} \mapsto x\}$ $\{[m, a, \text{if } m = \text{Own then } b \text{ else } -] \text{ emp}\}$ $\{[m, a, \text{if } m = \text{Own then } b \text{ else } -] x \mapsto v \wedge \text{res} = v\}$ $\{[m, a, \text{if } m = \text{Own then } b \text{ else } -] x \mapsto y\}$ $\{\exists b. [\text{Own}, a, b] I_* (a \bullet b)\}$ $\{[\text{NOwn}, \Phi(a), -] \text{ emp}\}$ $\{[m, a, \text{if } m = \text{Own then } b \text{ else } -] \text{ emp} \wedge \text{res} = x\}$
---	--	--	---

$$\frac{\Gamma \vdash \{p\}C_1 : B\{q\} \quad \Gamma, x:B \vdash \{[x/\text{res}]q\}C_2 : A\{r\} \quad x \notin \text{FV}(r)}{\Gamma \vdash \{p\}x \leftarrow C_1; C_2 : A\{r\}} \text{SEQ}$$

$$\frac{\Gamma \vdash \{p_1\}C_1 : A_1\{q_1\} \quad \Gamma \vdash \{p_2\}C_2 : A_2\{q_2\}}{\Gamma \vdash \{p_1 \otimes p_2\}C_1 \parallel C_2 : A_1 \times A_2 \{[\text{res.1}/\text{res}]q_1 \otimes [\text{res.2}/\text{res}]q_2\}} \text{PAR}$$

$$\frac{\Gamma \vdash \forall x:B. \{p\}F(x) : A\{q\} \quad \Gamma \vdash v : B}{\Gamma \vdash \{[v/x]p\}F(v) : A\{[v/x]q\}} \quad \frac{\Gamma, \forall x:B. \{p\}f(x) : A\{q\}, x:B \vdash \{p\}C : A\{q\}}{\Gamma \vdash \forall x:B. \{p\}(\text{fix } f.x.C)(x) : A\{q\}} \quad \frac{\forall x:B. \{p\}f(x) : A\{q\} \in \Gamma}{\Gamma \vdash \forall x:B. \{p\}f(x) : A\{q\}}$$

$$\frac{\Gamma \vdash \{e = \text{true} \wedge p\}C_1 : A\{q\} \quad \Gamma \vdash \{e = \text{false} \wedge p\}C_2 : A\{q\}}{\Gamma \vdash \{p\} \text{if } e \text{ then } C_1 \text{ else } C_2 : A\{q\}} \text{IF} \quad \frac{\Gamma \vdash \{p_1\}C : A\{q_1\} \quad \Gamma \vdash (p_1, q_1) \sqsubseteq (p_2, q_2)}{\Gamma \vdash \{p_2\}C : A\{q_2\}} \text{CONSEQ} \quad \frac{\Gamma \vdash \{p\}C : A\{q\} \quad x \notin \text{dom } \Gamma}{\Gamma \vdash \{\exists x:A. p\}C : A\{\exists x:A. q\}} \text{EXIST}$$

$$\frac{\Gamma \vdash \{p\}C : A\{q\}}{\Gamma \vdash \{p * r\}C : A\{q * r\}} \text{FRAME} \quad \frac{\Gamma \vdash \{p_1\}C : A\{q_1\} \quad \Gamma \vdash \{p_2\}C : A\{q_2\}}{\Gamma \vdash \{p_1 \wedge p_2\}C : A\{q_1 \wedge q_2\}} \text{CONJ}$$

Fig. 2. Subjective CSL rules $\Gamma \vdash \{p\}C : A\{q\}$ and $\Gamma \vdash \forall x:A_1. \{p\}F(x) : A_2\{q\}$.

Hence, in the postcondition, the environment contribution is specified via the conditional (if $m = \text{Own}$ then b else $-$).

b) Locking commands: The locking operations change the lock ownership status and perform the corresponding ownership transfer between the shared and local heaps.

To acquire the shared state, lock requires the thread to not own the lock ($m_S = \text{NOwn}$). Upon returning, the thread owns the lock ($m_S = \text{Own}$) and the thread-private heap has gained ownership of the shared resource satisfying $I_* (a_S \bullet a_O)$. The local contribution in the postcondition remains unchanged ($a_S = a$), but the environments contribution may have changed just prior to acquiring the lock. Thus, the new value for the environment contribution is an existentially abstracted b .

To release the resource, unlock requires the thread to own the lock ($m_S = \text{Own}$) and for the thread-private heap to satisfy the invariant $I_* (\Phi(a) \bullet b)$, relative to a *new* local contribution $\Phi(a)$ rather than the former local contribution a . Φ is a mathematical *local* function on U , passed to unlock to determine how the local auxiliary should be updated. When the command returns, the thread has lost ownership of the lock ($m_S = \text{NOwn}$) and the invariant heap has returned to the shared state. Moreover, the local contribution becomes $\Phi(a)$ and the environment contribution is unconstrained.

c) Parallel composition: The parallel composition $e_1 \parallel e_2$ forks into two children threads for e_1 and e_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 $p_1 \otimes p_2$ can be split into p_1 and p_2 , (2) each thread e_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 postcondition $q_1 \otimes q_2$, with the projections from the resulting pair substituted for the dedicated variable res . The precondition $p_1 \otimes p_2$ expresses that each of the initial values of h_S , m_S and a_S can be split

into two, so that p_1 and p_2 hold of the individual pieces of auxiliary state and the heap. We want to pass the pieces to two forked threads e_1 and e_2 , so we must ensure that p_1 and p_2 coherently describe the environments in which e_1 and e_2 run, respectively. Thus, if the parent thread has $(h_S, m_S, a_S, a_O) = (h_1 \bullet h_2, m_1 \bullet m_2, a_1 \bullet a_2, b)$, then the left child receives $(h_S, m_S, a_S, a_O) = (h_1, m_1, a_1, a_2 \bullet b)$ in its precondition p_1 , the reflecting that e_2 becomes part of e_1 's environment; a dual remark applies to the right child.

The parallel composition rule then states that if a partition satisfying the preconditions can be found of the initial world (encompassing h_S , m_S , a_S and a_O), then a partition satisfying the 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 e_1 and e_2 (and their common environment) in an interleaved manner, while updating their respective values of a_S and m_S . Whenever one thread's a_S is updated, the a_O 's of the other two threads must be updated correspondingly, in order to preserve the semantic relationship between the auxiliaries, that is, that the $a_S \bullet a_O$ for each thread yields the same collective contribution.

Global auxiliary state, as used in all the methods following Owicki and Gries, exposes the *concrete* environment contribution to all threads and thus the verification of a thread must track the partition between local and environment explicitly. Our insight is that the proof does not need to track the partition at all times; it only needs to know that one exists at the end, which satisfies the combined postconditions of the individual threads. Thus, subjective auxiliary state allows a thread to be verified in terms of its local contribution relative to an *abstract* environment contribution. The postcondition of parallel composition guarantees that there is a coherent way to combine the subthreads' local contributions, leaving some abstract, residual environment contribution.

d) *Other rules*: We support a combinator fix for general recursion, rather than just while loops. We have already illustrated it on the iterator example in Section II, which is not tail recursive, and thus difficult to write using only while loops. The inference rule requires proving a Hoare triple for the function body, under a hypothesis that the recursive calls already 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.

The FRAME rule allows splitting the heap into $p * r$, verifying the command with precondition p and postcondition q , then recombining the postcondition with the framed heap into $q * r$. Note that the rule uses $*$, not $\otimes$, so that unlike in PAR, the auxiliaries aren't framed. It seems possible to change the semantic definition of Hoare triples (given in Section V) to enable framing on the auxiliaries as well, but then the CONJ rule is unsound unless the $\bullet$ operation of U is also *cancellative*.⁵ As argued in Section III, the algebraic properties of $\bullet$ arise as abstractions of the properties of parallel composition, but cancellativity does not seem to, so we decided not to impose it on U . Somewhat surprisingly, the lack of framing on auxiliaries does not seem to influence the size of program proofs, nor their compositionality.

V. SEMANTICS AND SOUNDNESS

We start with the judgment $\Gamma \vdash (p_1, q_1) \sqsubseteq (p_2, q_2)$ that serves as a side condition on the rule of consequence in Figure 2. The semantic definition of this judgment (shown below) is closely tied to the treatment of logical variables.

The main idea is that in the presence of procedures, a logical variable should be interpreted locally to a Hoare triple. Given a triple $\{p\}C : A\{q\}$, a logical variable is implicitly universally quantified, to scope over p and q , but not over any other triple. Since the quantification is implicit, we determine the quantifiers out of the free variables of the assertions.

Let $\text{FLV}_\Gamma(r) \triangleq \text{FV}(r) \setminus (\text{dom } \Gamma \cup \{\text{res}\})$ be the set of logical variables of the assertion r . As a requirement for well-formedness of triples, we impose that $\text{FLV}_\Gamma(p) \supseteq \text{FLV}_\Gamma(q)$. The latter is needed for the soundness of the SEQ rule, to prevent the intermediate assertion from implicitly changing the number of semantic quantifiers when going from the conclusion to the premises. As logical variables are intended to relate p and q , it is not a real restriction.

Let $\bar{v}_i = \text{FLV}_\Gamma(p_i)$, for $i = 1, 2$. Then $\Gamma \vdash (p_1, q_1) \sqsubseteq (p_2, q_2)$ holds, iff for every substitution γ providing well typed expressions for variables in Γ , and for every two worlds w and w' , we have: (1) $w \models (\exists \bar{v}_2. \gamma p_2) \Rightarrow (\exists \bar{v}_1. \gamma p_1)$, and (2) $\forall \text{res} \in A. (\forall \bar{v}_1. w \models \gamma p_1 \Rightarrow w' \models \gamma q_1) \Rightarrow (\forall \bar{v}_2. w \models \gamma p_2 \Rightarrow w' \models \gamma q_2)$. Condition (1) states that p_1 can be strengthened into p_2 , and (2) states that q_1 can be weakened into q_2 . We make it explicit that p_1, p_2 scope over the initial world w of the computation, and q_1, q_2 scope over the ending

world w' . Quantification over $\bar{v}_1$ and $\bar{v}_2$ formally connects the two worlds. Notice that γ only concerns variables, but not Hoare triples that may appear in Γ . This suffices, as Hoare triples can't appear in assertions.

As this judgment is orthogonal to auxiliary state, we forgo further discussion of it, beyond showing that it leads to a sound rule of consequence. We do point out, however, that a similar formulation is necessary for a Hoare-style logic with procedures, as in that case, the customary rule of consequence is not strong enough. Our formulation closely follows the work of Kleymann [14, Th.3.2], which contains a broader analysis.⁶

The other judgments used in this section concern operational semantics. *Decorated* operational semantics $h; m; g; C \xrightarrow{\text{op}} h'; m'; g'; C'$ expresses the stepping of the command C wrt. the decorated state $h; m; g$. The *undecorated* semantics $h; C \xrightarrow{\text{op}} h'; C'$ omits the auxiliaries, and is thus the proper operational semantics. Both judgments assume that the dedicated lock of the logic is lk. Both are defined in the Appendix. Here, we just present the following lemma expressing their relationship. The lemma is important because it shows that auxiliary state doesn't have run-time effects (e.g., an auxiliary can't be copied into a heap pointer). It is a standard lemma for any Hoare logic that uses auxiliary state.

Lemma 1 (Erasure). *For every h, m, g , there exists h', m', g' such that $h; m; g; C \xrightarrow{\text{op}} h'; m'; g'; C'$ in the decorated semantics, iff $h; C \xrightarrow{\text{op}} h'; C'$ in the undecorated one.*

As a first step in the proof of soundness of SCSL, we establish the substitution principles for the two Hoare triple judgments (ranged over by J). The proofs are by induction on the derivation of J . These lemmas are important, as they establish the compositionality of SCSL.

Lemma 2 (Substitution for expression variables). *If $\Gamma \vdash e : A$ and $\Gamma, x:A, \Gamma' \vdash J$, then $\Gamma, [e/x] \Gamma' \vdash [e/x] J$.*

Lemma 3 (Substitution for function variables). *If $\Gamma \vdash \forall x : A. \{p\} F(x) : B\{q\}$ and $\Gamma, \forall x:A. \{p\} f(x) : B\{q\}, \Gamma' \vdash J$, then $\Gamma, \Gamma' \vdash [F/f] J$.*

Next, we require several helper definitions which we use to relate Hoare triples to the operational semantics.

Definition 2 (Precision). *A predicate I over heaps is precise if it circumscribes a unique subheap for every heap h . That is, for all h_1, h_2 subheaps of h , if $I h_1$ and $I h_2$ then $h_1 = h_2$.*

Definition 3 (Configuration). *Let U be a PCM of user-defined auxiliary state, lk be a lock (i.e., boolean pointer), and I_* be a predicate over U and heap such that $I_*(\alpha)$ is precise for all $\alpha \in U$. The 7-tuple $(h_S; m_S; a_S \mid h_R \mid h_O; m_O; a_O)$, where $h_S, h_R, h_O \in \text{heap}$, $m_S, m_O \in \text{owns}$ and $a_S, a_O \in U$ is a configuration, ranged over by c , if the following holds.*

- 1) valid $(h_S \uplus h_R \uplus h_O)$, valid $(m_S \bullet m_O)$, valid $(a_S \bullet a_O)$
- 2) $m_S \bullet m_O = \text{NOwn} \Rightarrow \exists h. h_R = \text{lk} \rightsquigarrow \text{false} \uplus h \wedge I_*(a_S \bullet a_O) h$
- 3) $m_S \bullet m_O = \text{Own} \Rightarrow h_R = \text{lk} \rightsquigarrow \text{true}$

⁵Cancellativity means that if $x \bullet y = x \bullet z$, then $y = z$; a property that heap, as a special PCM, does satisfy.

⁶Incidentally, in *loc. cit.*, logical variables are called auxiliary variables, but are unrelated to our auxiliary state.

Given a thread t , the configuration c represents the subjective view that t may have of the global (heap and auxiliary) state. In SCSL, the heap can be logically partitioned from the point of view of the thread being specified: a heaplet h_S that is local to the thread, a heaplet h_O local to the environment (which may be further partitioned, if the environment thread forks), and a shared heaplet h_R storing a lock lk and the shared resource that the lock protects, and which is required to satisfy the invariant I_* . This partitioning is merely logical; it is apparent in the inference rules, but has no counterpart in the operational semantics. Similarly, the lock status and the auxiliary state can be split into m_S, m_O and a_S, a_O .

As in CSL, the shared resource transfers into h_S upon the thread acquiring the lock, after which it can be modified, and the invariant I_* violated. When unlocking, however, I_* must be re-established, and the part of h_S satisfying I is transferred back to h_R . In order to uniquely determine the part to transfer, the predicate $I_*(\alpha)$ is required to be *precise* in the sense of Definition 2.

When either t or its environment owns lk ($m_S \bullet m_O = \text{Own}$), then basic coherence demands that lk must be set to true. This requirement is specified by condition (3) in Definition 3. If neither t nor the environment owns lk , then lk must be set to false, but the shared heap contains an additional chunk h , identified by the invariant I_* , as specified by condition (2). The condition (1) insists that all the components of a configuration combine in a valid way; otherwise, the configuration is not reachable by programs, and need not be considered.

Definition 4. Given a configuration $c = (h_S; m_S; a_S \mid h_R \mid h_O; m_O; a_O)$, we define c 's flattening $\llbracket c \rrbracket$, c 's heap $[c]$, c 's world $\llbracket c \rrbracket$, and c 's transpose $c^\dagger$, as follows.

$$\begin{aligned}\llbracket c \rrbracket &\triangleq h_S \uplus h_R \uplus h_O; m_S \bullet m_O; a_S \bullet a_O \\ [c] &\triangleq h_S \uplus h_R \uplus h_O \\ \llbracket c \rrbracket &\triangleq h_S; m_S; a_S; a_O \\ c^\dagger &\triangleq (h_O; m_O; a_O \mid h_R \mid h_S; m_S; a_S)\end{aligned}$$

In the following definition, we assume that a *decorated state*, ranged over by d , is a triple of the form $h; m; a$.

Definition 5. A configuration transition, or simply transition, is a pair of configurations (c, c') . A transition is a local transition, written $c \rightarrow_S c'$, if it has one of the following forms:

$$\begin{aligned}\text{private} : & \quad h_S; m_S; a_S \mid h_R \mid d_O \rightarrow_S h'_S; m_S; a_S \mid h_R \mid d_O \\ \text{acquire} : & \quad h_S; \text{NOwn}; a_S \mid lk \rightsquigarrow \text{false} \uplus h_{\text{inv}} \mid d_O \rightarrow_S \\ & \quad h_S \uplus h_{\text{inv}}; \text{Own}; a_S \mid lk \rightsquigarrow \text{true} \mid d_O \\ \text{release} : & \quad h_S \uplus h_{\text{inv}}; \text{Own}; a_S \mid lk \rightsquigarrow \text{true} \mid d_O \rightarrow_S \\ & \quad h_S; \text{NOwn}; a'_S \mid lk \rightsquigarrow \text{false} \uplus h_{\text{inv}} \mid d_O\end{aligned}$$

A transition is an environment transition, written $c \rightarrow_O c'$, iff $c^\dagger \rightarrow_S c'^\dagger$.

The three kinds of local transitions correspond to the changes that the commands of SCSL may perform. A command may just change the local heap (private transition),

acquire the lock and gain access to the shared heap (acquire transition), or release the lock and lose access to the shared heap (release transition). Notice that while private and acquire transitions preserve validity of configurations, the release transition by itself may not: the transferred subheap h_{inv} may not satisfy the invariant $I_*(a'_S \bullet a_O)$. We will enforce that even release transitions preserve validity by encoding such a requirement into a predicate *always*, defined below, that iterates configuration transitions.

Our Hoare triples are *fault avoiding*; every verified command is safe to execute in a heap satisfying the precondition in the sense that it will not cause memory faults or type errors. To encode fault avoidance into the semantics of Hoare triples, we require a heap predicate *safe_st* C describing *safety for one step* for the command C . Eventually, we will iterate one-step safety to encode safety for arbitrary many steps.

$$\begin{aligned}\text{safe_st}(\text{alloc } e) h &\triangleq \text{True} \\ \text{safe_st}(\text{dealloc } e) h &\triangleq e \in \text{dom } h \\ \text{safe_st}(\text{read}_A e) h &\triangleq \exists v \in A. h = e \rightsquigarrow v \uplus h' \\ \text{safe_st}(\text{write } e \ e') h &\triangleq h = e \rightsquigarrow - \uplus h' \\ \text{safe_st}(\text{lock}) h &\triangleq \exists v \in \text{bool}. h = lk \rightsquigarrow v \uplus h' \\ \text{safe_st}(\text{unlock}_\Phi) h &\triangleq \exists v \in \text{bool}. h = lk \rightsquigarrow v \uplus h' \\ \text{safe_st}(\text{ret } e) h &\triangleq \text{True} \\ \text{safe_st}(F(e)) h &\triangleq \text{True} \\ \text{safe_st}(x \leftarrow C_1; C_2) h &\triangleq \text{safe_st } C_1 h \\ \text{safe_st}(C_1 \parallel C_2) h &\triangleq \text{safe_st } C_1 h \wedge \text{safe_st } C_2 h \\ \text{safe_st}(\text{if } e \text{ then } C_1 \text{ else } C_2) h &\triangleq \text{True}\end{aligned}$$

In the case of primitive memory and synchronization commands, *safe_st* simply captures the existence of the pointer that the command dereferences. Function application $F(e)$ is always memory safe for one step, because its first step merely involves unfolding F , and substituting e . One-step safety of sequential composition requires one-step safety of the first command, and for parallel composition, of both commands.

Next, we define the *always_O* and *always_S* predicates by induction on the number n of operational steps. Both are indexed by n and have three arguments: a configuration c , an A -returning command C , and a predicate K over a configuration and an A -returning command. The definitions encode that the command C is safe to run for n steps in parallel with an arbitrary environment starting from c , and that K holds of the thread and the configuration at every point during the execution.

$$\begin{aligned}\text{always}_O^n c C K &\triangleq \forall c'. (\text{valid } c' \wedge c \rightarrow_O^* c') \Rightarrow \text{always}_S^n c' C K \\ \text{always}_S^n c C K &\triangleq \text{safe_st } C [c] \wedge K c C \wedge \\ &\quad \forall n' d' C'. (n' < n \wedge \llbracket c \rrbracket; C \xrightarrow{O} d'; C') \Rightarrow \\ &\quad \exists c'. \llbracket c' \rrbracket = d' \wedge \text{valid } c' \wedge c \rightarrow_S c' \wedge \text{always}_O^{n'} c' C' K\end{aligned}$$

always_O simulates arbitrary environment interference by a transition from c to a valid c' , by any number of environment transitions; then it recurses to give the thread a turn. *always_S* ensures that the command C and configuration c are safe for one step, and satisfy predicate K . Moreover, for any step of $\llbracket c \rrbracket; C$ to $d'; C'$, we require that the resulting decorated state d' can be viewed as a flattening of some valid configuration

c' . That is, it should be possible to split the decorated state d' into local, shared and environment partitions in a coherent way, thus ensuring that the operational step from $\llbracket c \rrbracket$ to d' corresponds to a local transition from c to c' . This check for existence of c' enforces the mutual exclusion protocol. First, the auxiliary for lock ownership should split in a coherent way, so it is impossible for both the local thread and the environment thread to own the lock (that is, $m_S = m_O = \text{Owns}$). Second, since we require that c' is a valid configuration, we enforce that upon lock release, the resource invariant holds of the shared heap. After the check for existence of c' , always_S recurses to give the environment more turns. Note how the step index n decreases after each time the thread gets a turn, thus resulting in a well-founded definition.

We next restrict always to express properties of the ending state of a command. Given a predicate Q over configurations and A -values, $\text{after}^n c C Q$ holds iff C is safe for n steps, and if C terminates in n steps, then Q holds of the ending configuration and value.

$$\text{after}^n c C Q \triangleq \text{always}_O^n c C (\lambda c' C'. \forall v' \in A. C' = \text{ret } v' \Rightarrow Q c' v')$$

Lemma 4 (Determinism). *If $\llbracket c \rrbracket; C \xrightarrow{\text{op}} d'; C'$, $\llbracket c'_1 \rrbracket = d' = \llbracket c'_2 \rrbracket$, $c \rightarrow_S c'_1$, and $c \rightarrow_S c'_2$, then $c'_1 = c'_2$.*

Proof: By induction on the command stepping. ■

Lemma 5 (Universals). *If $\text{always}_O^n c C (\lambda c' C'. \text{True})$, then $\text{always}_O^n c C (\lambda c' C'. \forall x \in A. p \ x \ c' \ C')$ iff for all $x \in A$, $\text{always}_O^n c C (\lambda c' C'. p \ x \ c' \ C')$.*

Proof: By induction on n . ■

Lemma 6 (Normality). *If $\text{after}^n c C Q_1$ and for all $c', v' \in A$, $Q_1 c' v' \Rightarrow Q_2 c' v'$, then $\text{after}^n c C Q_2$.*

Proof: By induction on n . ■

Lemma 7 (Closure under sequential composition). *If $\text{after}^n c C_1 (\lambda c' x. \forall m. \text{after}^m c' C_2 Q)$, then $\text{after}^n c (\text{let } x = C_1 \text{ in } C_2) Q$.*

Proof: By induction on n . ■

Lemma 8 (Closure under parallel composition).

If $\text{after}^n (d_1 \mid h_R \mid d_2 \bullet d_O) C_1 Q_1$ and $\text{after}^n (d_2 \mid h_R \mid d_1 \bullet d_O) C_2 Q_2$, then $\text{after}^n (d_1 \bullet d_2 \mid h_R \mid d_O) (C_1 \parallel C_2) (Q_1 \otimes Q_2)$, where $\otimes$ on predicates generalizes $\otimes$ on assertions, as follows:

$$Q_1 \otimes Q_2 \triangleq \lambda c' v'. \exists d'_1 d'_2 h'_R d'_O. c' = (d'_1 \bullet d'_2 \mid h'_R \mid d'_O) \wedge \\ Q_1 (d'_1 \mid h'_R \mid d'_2 \bullet d'_O) (v'.1) \wedge \\ Q_2 (d'_2 \mid h'_R \mid d'_1 \bullet d'_O) (v'.2)$$

Proof: By induction on n . Intuitively, when C_1 takes a step it is environment interference on C_2 , and vice versa. ■

We are now ready to define the meaning of the Hoare triples,

in the case the context Γ is empty, for n steps of the command.

$$\begin{aligned} \llbracket \{p\} C : A \{q\} \rrbracket_n &\triangleq \\ &\forall \text{FLV}(p, q) \ c \ h. \llbracket c \rrbracket \models p * \text{this}(h) \Rightarrow \\ &\quad \text{after}^n c C (\lambda c' \text{res}. \llbracket c' \rrbracket \models q * \text{this}(h)) \\ \llbracket \forall x:B. \{p\} F(x) : A \{q\} \rrbracket_n &\triangleq \forall x \in B. \llbracket \{p\} F(x) : A \{q\} \rrbracket_n \end{aligned}$$

Intuitively, C has a precondition p and postcondition q iff: for any initial configuration c whose world $\llbracket c \rrbracket$ satisfies the precondition p : (1) C is safe to execute in the sense that it does not dereference inexistent or ill-typed memory cells (as definition of always includes an safe_st conjunct); (2) C respects mutual exclusion when executed against an environment that respects mutual exclusion (follows from recursive definitions of always_O and always_S). In particular, the environment can non-deterministically acquire or release the lock whenever C is outside the critical section, and have the guarantee that resource invariant I ($a_S \bullet a_O$) holds, i.e., the shared heap and a_O changed in tandem; (3) q holds of the ending world and result (expressed by the predicate passed to after).

The definition quantifies over all logical variables of p and q , in order to give these variables local scope. Additionally, we follow some earlier work on separation logic [15], [16] and embed the small footprint semantics into the Hoare triples by explicitly quantifying over the heap h that C does not touch. We assert that h is invariant by means of Reynolds' predicate $\text{this}(h)$ [17], defined by:

$$h_S; m_S; a_S; a_O \models \text{this}(h) \text{ iff } h_S = h$$

Finally, the semantics of the hypothetical Hoare judgments is defined by quantifying over the number of steps n , and substitutions γ that are appropriate for n steps.

$$\llbracket \Gamma \vdash J \rrbracket \triangleq \forall n \ \gamma. \gamma \models_n \Gamma \Rightarrow \llbracket \gamma \ J \rrbracket_n$$

where

$$\begin{aligned} \gamma \models_n \cdot &\text{ iff true} \\ \gamma \models_n x:A, \Gamma' &\text{ iff } \gamma(x) = e, \cdot \vdash e : A, \\ &\text{ and } \gamma \models_n [e/x] \Gamma' \\ \gamma \models_n \forall x:B. \{p\} f(x) : A \{q\}, \Gamma' &\text{ iff } \gamma(f) = F, \\ &\text{ and } \llbracket \forall x:B. \{p\} F(x) : A \{q\} \rrbracket_n, \\ &\text{ and } \gamma \models_n \Gamma' \end{aligned}$$

Theorem 1 (Soundness). *If $\Gamma \vdash J$, then $\llbracket \Gamma \vdash J \rrbracket$.*

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 safe_st , and the resulting configuration satisfies the postcondition. The sequential and parallel rules are sound by the respective closure lemmas. The fix rule is sound by an inner induction on the number of steps n . The conjunction rule is sound by the universal lemma and cancellativity on heaps. The consequence, existential, and frame rules are sound by logical deduction. ■

VI. RELATED WORK

RGSep [6] and SAGL [7], [18] combine Jones' rely/guarantee [4] with separation logic. Rely/guarantee improves on the resource invariants method because it distinguishes thread and environment interference, which can avoid auxiliary state compared to resource invariant proofs. However, sufficiently complex interference (e.g., incrementing in parallel) still requires noncompositional auxiliary state in rely/guarantee proofs.

The HLRG logic [19] combines rely/guarantee and temporal reasoning over explicit history traces, which eliminates the need for auxiliary state. For examples such as incrementing in parallel, reasoning over traces must consider all interleavings, which grows exponentially in the number of threads and thus fails to be compositional. HLRG only handles one top-level parallel composition of a fixed number of threads, whereas SCSL allows parallel composition in any command and the number of threads can vary dynamically.

Resource invariants and rely/guarantee allow verifying a thread independently of its context, except when specifying thread interactions requires auxiliary state. Likewise, recent work on concurrent abstract predicates and fine-grained concurrency specifications allow verifying a procedure independently of its call sites. However, they still use (objective) auxiliary state which makes their proofs sensitive to the names and number of threads, and thus cannot easily support our iterator example.

A concurrent abstract predicate [8] serves to decouple verification of a module in terms of a data structure's concrete state from verification of the client in terms of the abstract state. While the approach allows showing that multiple modules satisfy the same specification, multiple clients of a single module may require it to expose different abstract predicates and different specifications. In particular, proving the two- and three-thread clients of increment would require increment to provide different abstract predicates.

Fine-grained concurrency specifications [9] serve to verify a module parametrically in the client's auxiliary state and invariant; then clients can instantiate the module with particular auxiliary state, invariant, and auxiliary update code as needed. This allows verifying the standalone increment program once, but then the two- and three-thread clients of increment need to instantiate it with different auxiliary state and invariants. Thus, even though the two-thread program is contained in the three-thread program, its proof can't be reused.

VII. CONCLUSION

We propose Subjective Concurrent Separation Logic (SCSL) as a combination of the resource invariants method and separation logic together with *subjective* auxiliary state, which overcomes the difficulties of standard (objective) auxiliary state. Each thread has a subjective perspective that splits auxiliary state into contributions by the *self* (the thread itself) and the *other* (its environment), which makes proofs insensitive to the internal concurrency (i.e., number of threads) of the environment. The approach is enabled by recognizing that

auxiliary contributions have the algebraic structure of a partial commutative monoid.

SCSL yields *compositional* proofs in the sense that a program can be proved once and the proof can be reused directly in different contexts. For the classical example of incrementing in parallel, we can give a compositional proof that works independently of the number of parallel instances, whereas existing logics need a different global proof for each particular forking pattern. Moreover, we show how subjective auxiliary state enables verification of a concurrent iterator parametrically in the list.

In future work, we will extend the language from a first-order to a higher-order language with orthogonal language features such as multiple locks, and dynamic lock and thread creation, and extend the logic from resource invariants to rely/guarantee interference specifications. We will also mechanize the metatheory in the Coq proof assistant.

REFERENCES

- [1] C. B. Jones, "The role of auxiliary variables in the formal development of concurrent programs," University of Newcastle upon Tyne, Computing Science, Tech. Rep. CS-TR-1179, 2009.
- [2] S. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Inf.*, vol. 6, 1976.
- [3] —, "Verifying properties of parallel programs: An axiomatic approach," *Commun. ACM*, vol. 19, no. 5, 1976.
- [4] C. B. Jones, "Specification and design of (parallel) programs," in *IFIP Congress*, 1983.
- [5] P. W. O'Hearn, "Resources, concurrency, and local reasoning," *Theor. Comput. Sci.*, vol. 375, no. 1-3, 2007.
- [6] V. Vafeiadis and M. J. Parkinson, "A marriage of rely/guarantee and separation logic," in *CONCUR'07*.
- [7] X. Feng, R. Ferreira, and Z. Shao, "On the relationship between concurrent separation logic and assume-guarantee reasoning," in *ESOP'07*.
- [8] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis, "Concurrent abstract predicates," in *ECOOP'10*.
- [9] B. Jacobs and F. Piessens, "Expressive modular fine-grained concurrency specification," in *POPL'11*.
- [10] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson, "Permission accounting in separation logic," in *POPL*, 2005.
- [11] R. Bornat, C. Calcagno, and H. Yang, "Variables as resource in separation logic," *ENTCS*, vol. 155, 2006.
- [12] M. J. Parkinson, R. Bornat, and C. Calcagno, "Variables as resource in Hoare logics," in *LICS'06*.
- [13] U. S. Reddy and J. C. Reynolds, "Syntactic control of interference for separation logic," in *POPL'12*.
- [14] T. Kleymann, "Hoare logic and auxiliary variables," *Formal Aspects of Computing*, vol. 11, 1999.
- [15] L. Birkedal and H. Yang, "Relational parametricity and separation logic," in *FOSSACS'07*, ser. LNCS, vol. 4423, 2007.
- [16] A. Nanevski, V. Vafeiadis, and J. Berdine, "Structuring the verification of heap-manipulating programs," in *POPL*, 2010.
- [17] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *LICS*, 2002.
- [18] X. Feng, "Local rely-guarantee reasoning," in *POPL'09*.
- [19] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang, "Reasoning about optimistic concurrency using a program logic for history," in *CONCUR*, 2010.

APPENDIX

$A ::= \text{unit} \mid \text{bool} \mid \text{nat} \mid \text{ptr} \mid A_1 \times A_2$	expression types
$e ::= x \mid () \mid \text{true} \mid 0 \mid \ell \mid (e_1, e_2) \mid \dots$	expressions
$C ::= \text{alloc } e \mid \text{dealloc } e \mid \text{read } e \mid \text{write } e_1 \ e_2$ $\mid \text{unlock}_{\Phi} \mid \text{lock} \mid \text{ret } e \mid x \leftarrow C_1; C_2 \mid C_1 \parallel C_2$ $\mid F(e) \mid \text{if } e \text{ then } C_1 \text{ else } C_2$	commands
$F ::= f \mid \text{fix } f.x.C$	command functions

Fig. 3. Language syntax.

$h_S; m_S; a_S; a_O \models \text{emp}$	iff $h_S = \text{empty}$
$h_S; m_S; a_S; a_O \models p_1 * p_2$	iff $h_1; m_S; a_S; a_O \models p_1$ and $h_2; m_S; a_S; a_O \models p_2$ for some h_1 and h_2 s.t. $h_S = h_1 \bullet h_2$
$h_S; m_S; a_S; a_O \models x \mapsto v$	iff $h_S = x \rightsquigarrow v$
$h_S; m_S; a_S; a_O \models [m, a_1, a_2] p$	iff $(m_S, a_S, a_O) = (m, a_1, a_2)$ and $h_S; m; a_1; a_2 \models p$
$h_S; m_S; a_S; a_O \models p_1 \circledast p_2$	iff $h_1; m_1; a_1; a_2 \bullet a_O \models p_1$ and $h_2; m_2; a_2; a_1 \bullet a_O \models p_2$ for some $h_1, h_2, m_1, m_2, a_1, a_2$ s.t. $h_S = h_1 \bullet h_2, m_S = m_1 \bullet m_2, a_S = a_1 \bullet a_2$
$h_S; m_S; a_S; a_O \models \text{True}$	iff true
$h_S; m_S; a_S; a_O \models p_1 \wedge p_2$	iff $h_S; m_S; a_S; a_O \models p_1$ and $h_S; m_S; a_S; a_O \models p_2$
$h_S; m_S; a_S; a_O \models p_1 \vee p_2$	iff $h_S; m_S; a_S; a_O \models p_1$ or $h_S; m_S; a_S; a_O \models p_2$
$h_S; m_S; a_S; a_O \models p_1 \Rightarrow p_2$	iff $h_S; m_S; a_S; a_O \models p_1$ implies $h_S; m_S; a_S; a_O \models p_2$
$h_S; m_S; a_S; a_O \models \forall x:A.p$	iff $h_S; m_S; a_S; a_O \models [v/x]p$ for all $v : A$
$h_S; m_S; a_S; a_O \models \exists x:A.p$	iff $h_S; m_S; a_S; a_O \models [v/x]p$ for some $v : A$
$h_S; m_S; a_S; a_O \models e_1 = e_2$	iff $e_1 = e_2$

Fig. 5. Semantics of assertions $h_S; m_S; a_S; a_O \models p$.

$$\begin{array}{c}
\frac{\ell \notin \text{dom } h}{h; m; a; \text{alloc } e \xrightarrow{\text{op}} h \bullet \ell \rightsquigarrow e; m; a; \text{ret } \ell} \quad \frac{h = h' \bullet e \rightsquigarrow -}{h; m; a; \text{dealloc } e \xrightarrow{\text{op}} h'; m; a; \text{ret } ()} \\
\\
\frac{h = - \bullet e \rightsquigarrow v'}{h; m; a; \text{read } e \xrightarrow{\text{op}} h; m; a; \text{ret } v'} \quad \frac{h = h' \bullet e \rightsquigarrow -}{h; m; a; \text{write } e \xrightarrow{\text{op}} h' \bullet e \rightsquigarrow e'; m; a; \text{ret } ()} \\
\\
\frac{h = h' \bullet \text{lk} \rightsquigarrow - \quad \Phi(a) = a'}{h; m; a; \text{unlock}_\Phi \xrightarrow{\text{op}} h' \bullet \text{lk} \rightsquigarrow \text{false}; \text{NOwn}; a'; \text{ret } ()} \quad \frac{h = h' \bullet \text{lk} \rightsquigarrow \text{false}}{h; m; a; \text{lock} \xrightarrow{\text{op}} h' \bullet \text{lk} \rightsquigarrow \text{true}; \text{Own}; a; \text{ret } ()} \quad \frac{h = h' \bullet \text{lk} \rightsquigarrow \text{true}}{h; m; a; \text{lock} \xrightarrow{\text{op}} h; m; a; \text{lock}} \\
\\
\frac{}{h; m; a; x \leftarrow \text{ret } e; C_2 \xrightarrow{\text{op}} h; m; a; [e/x]C_2} \quad \frac{h; m; a; C_1 \xrightarrow{\text{op}} h'; m'; a'; C'_1}{h; m; a; x \leftarrow C_1; C_2 \xrightarrow{\text{op}} h'; m'; a'; x \leftarrow C'_1; C_2} \\
\\
\frac{h; m; a; C_1 \xrightarrow{\text{op}} h'; m'; a'; C'_1}{h; m; a; C_1 \parallel C_2 \xrightarrow{\text{op}} h'; m'; a'; C'_1 \parallel C_2} \quad \frac{h; m; a; C_2 \xrightarrow{\text{op}} h'; m'; a'; C'_2}{h; m; a; C_1 \parallel C_2 \xrightarrow{\text{op}} h'; m'; a'; C_1 \parallel C'_2} \quad \frac{}{h; m; a; \text{ret } v_1 \parallel \text{ret } v_2 \xrightarrow{\text{op}} h; m; a; \text{ret } (v_1, v_2)} \\
\\
\frac{}{h; m; a; (\text{fix } f.x.C)e \xrightarrow{\text{op}} h; m; a; [\text{fix } f.x.C/f][e/x]C} \quad \frac{e = \text{true}}{h; m; a; \text{if } e \text{ then } C_1 \text{ else } C_2 \xrightarrow{\text{op}} h; m; a; C_1} \quad \frac{e = \text{false}}{h; m; a; \text{if } e \text{ then } C_1 \text{ else } C_2 \xrightarrow{\text{op}} h; m; a; C_2}
\end{array}$$

Fig. 4. Decorated operational stepping for commands $h; m; a; C \xrightarrow{\text{op}} h'; m'; a'; C'$ (undecorated stepping is obtained by dropping m, a, m' , and a').