Barriers in Concurrent Separation Logic

Aquinas Hobor and Cristian Gherghina

National University of Singapore

Abstract. We develop and prove sound a concurrent separation logic for Pthreads-style barriers. Although Pthreads barriers are widely used in systems, and separation logic is widely used for verification, there has not been any effort to combine the two. Unlike locks and critical sections, Pthreads barriers enable simultaneous resource redistribution between multiple threads and are inherently stateful, leading to significant complications in the design of the logic and its soundness proof. We show how our logic can be applied to a specific example program in a modular way. Our proofs are machine-checked in Coq.

1 Introduction

In a shared-memory concurrent program, threads communicate via a common memory. Programmers use synchronization mechanisms, such as critical sections and locks, to avoid data races. In a data race, threads "step on each others' toes" by using the shared memory in an unsafe manner. Recently, concurrent separation logic has been used to formally reason about shared-memory programs that use critical sections and (first-class) locks [18, 15, 13, 14]. Programs verified with concurrent separation logic are provably data-race free.

What about shared-memory programs that use other kinds of synchronization mechanisms, such as semaphores? The general assumption is that other mechanisms can be implemented with locks, and that reasonable Hoare rules can be derived by verifying their implementation. Indeed, the first published example of concurrent separation logic was implementing semaphores using critical sections [18]. Unfortunately, not all synchronization mechanisms can be easily reduced to locks in a way that allows for a reasonable Hoare rule to be derived. In this paper we introduce a Hoare rule that natively handles one such synchronization mechanism, the Pthreads-style barrier.

Pthreads (POSIX Threads) is a widely-used API for concurrent programming, and includes various procedures for thread creation/destruction and synchronization [7]. When a thread issues a barrier call it waits until a specified number (typically all) of other threads have also issued a barrier call; at that point, all of the threads continue. Although barriers do not get much attention in theory-oriented literature, they are very common in actual systems code. PARSEC is the standard benchmarking suite for multicore architectures, and has thirteen workloads selected to provide a realistic cross-section for how concurrency is used in practice today; a total of five (38%) of PARSEC's workloads use barriers, covering the application domains of financial analysis (blackscholes),

computer vision (bodytrack), engineering (canneal), animation (fluidanimate), and data mining (streamcluster) [4]. A common use for barriers is to manage large numbers of threads in a pipeline setting. For example, in a video-processing algorithm, each thread might read from some shared common area containing the most recently completed frame while writing to some private area that will contain some fraction of the next frame. (A thread might need to know what is happening in other areas of the previous frame to properly handle objects entering or exiting its part of the current frame.) In the next iteration, the old private areas become the new shared common area as the algorithm continues.

Our key insight is that a barrier is used to simultaneously redistribute ownership of resources (typically, permission to read/write memory cells) between multiple threads. In the video-processing example, each thread starts out with read-only access to the previous frame and write access to a portion of the current frame. At the barrier call, each thread gives up its write access to its portion of the (just-finished) frame, and receives back read-only access to the entire frame. Separation logic (when combined with fractional permissions [5, 11]) can elegantly model this kind of resource redistribution. Let Pre_i be the preconditions held upon entering the barrier, and $Post_i$ be the postconditions that will hold after being released; then the following equation is *almost* true:

$$\mathop{\Large\ast}_{i} Pre_i \quad = \quad \mathop{\Large\ast}_{i} Post_i \tag{1}$$

Pipelined algorithms often operate in stages. Since barriers are used to ensure that one computation has finished before the next can start, the barriers need to have stages as well—a piece of ghost state associated with the barrier. We model this by building a finite automata into the barrier definition. We then need an assertion, written $\mathsf{barrier}(bn, \pi, cs)$, which says that barrier bn, owned with fractional permission π, is currently in state cs. The state of a barrier changes exactly as the threads are released from the barrier. We can correct equation (1) by noting that barrier bn is transitioning from state cs (current state) to state ns (next state), and that the other resources (frame F) are not modified:

$$\begin{array}{lcl} \mathop{\Large\ast}_{i} Pre_i & = & F * \mathsf{barrier}(bn, \blacksquare, cs) \\ \mathop{\Large\ast}_{i} Post_i & = & F * \mathsf{barrier}(bn, \blacksquare, ns) \end{array} \tag{2}$$

We use the symbol $\blacksquare$ to denote the full ($\sim$100%) permission, which we require so that no thread has a "stale" view of the barrier state. Although the on-chip (or *erased*) operational behavior of a barrier is conceptually simple[1], it may be already apparent that the verification can rapidly become quite complicated.

Contributions.

1. We give a formal characterization for sound barrier definitions.
2. We design a natural Hoare rule in separation logic for verifying barrier calls.
3. We give a formal resource-aware *unerased* concurrent operational semantics for barriers and prove our Hoare rules sound with respect to our semantics.
4. Our soundness results are machine-checked in Coq and are available at:

`www.comp.nus.edu.sg/~hobor/barrier`

[1] Suspend each thread as it arrives; keep a counter of the number of arrived threads; and when all of the threads have arrived, resume the suspended threads.

2 Syntax, Separation Algebras, Shares, and Assertions

Here we briefly introduce preliminaries: the syntax of our language, separation algebras, share accounting, and the assertions of our separation logic.

2.1 Programming Language Syntax

To let us focus on the barriers, most of our programming language is pure vanilla. We define four kinds of (tagged) values v: `TRUE`, `FALSE`, `ADDR`($\mathbb{N}$), and `DATA`($\mathbb{N}$). We have two (tagged) expressions e: `C`(v) and `V`(x), where x are local variable names (just $\mathbb{N}$ in Coq). To make the example more interesting we add the arithmetical operations to e. We write `bn` for a barrier number, with `bn` $\in \mathbb{N}$.

We have ten commands c: `skip` (do nothing), $x := e$ (local variable assignment), $x := [e]$ (load from memory), $[e_1] := e_2$ (store to memory), x:= `new` e (memory allocation), `free` e (memory deallocation), c_1; c_2 (instruction sequence), `if` e `then` c_1 `else` c_2 (if-then-else), `while` e $\{c\}$ (loops), and **barrier** `bn` (wait for barrier `bn`). To run commands $c_1 \ldots c_n$ in parallel (which, like O'Hearn, we only allow at the top level [18]), we write $c_1 || \ldots || c_n$. To avoid clogging the presentation, we elide a setup sequence before the parallel composition.

2.2 Disjoint Multi-unit Separation Algebras

Separation algebras are mathematical structures used to model separation logic. We use a variant described by Dockins *et al.* called a disjoint multi-unit separation algebra (hereafter just "DSA") [11]. Briefly, a DSA is a set S and an associated three-place partial *join relation* $\oplus$, written $x \oplus y = z$, such that:

A function:	$x \oplus y = z_1 \;\Rightarrow\; x \oplus y = z_2 \;\Rightarrow\; z_1 = z_2$
Commutative:	$x \oplus y \;=\; y \oplus x$
Associative:	$x \oplus (y \oplus z) \;=\; (x \oplus y) \oplus z$
Cancellative:	$x_1 \oplus y = z \;\Rightarrow\; x_2 \oplus y = z \;\Rightarrow\; x_1 = x_2$
Multiple units:	$\forall x.\ \exists u_x.\ x \oplus u_x = x$
Disjointness:	$x \oplus x = y \;\Rightarrow\; x = y$

A key concept is the idea of an *identity*: x is an identity if $x \oplus y = z$ implies $y = z$. One fundamental property of identities is that x is an identity if and only if $x \oplus x = x$. Dockins also develops a series of standard constructions (*e.g.*, product, functions, etc.) for building complicated DSAs from simpler DSAs. We make use of this idea to construct a variety of separation algebras as needed, usually with the concept of *share* as the "foundational" DSA.

2.3 Shares

Separation logic is a logic of *resource ownership*. Concurrent algorithms sometimes want to have threads share some common resources. Bornat *et al.* introduced the concept of *fractional share* to handle the necessary accounting [5]. Shares form a DSA; a *full share* (complete ownership of a resource) can be broken into various *partial shares*; these shares can then be rejoined into the full share. The *empty share* is the identity for shares. We often need non-empty

(strictly *positive*) shares, denoted by π. A critical invariant is that the sum of each thread's share of a given object is no more or less than the full share.

The semantic meaning of partial shares varies; here we use them in two distinct ways. We require the full share to modify a memory location; in contrast, we only require a positive share to read from one. There is no danger of a data race even though we do not require the full share to read: if a thread has a positive share of some location, no other thread can have a full share for the same location. We use fractional permissions differently for barriers: each precondition includes some positive share of the barrier itself and we require that the preconditions combine to imply the full share of the barrier (plus a frame F).

In the Coq development we use a share model developed Dockins *et al.* that supports sophisticated fractional ownership schemes [11]. Here we simplify this model into four elements: the full share ■; two **distinct** nonempty partial shares, ◩ and ◪, and the empty share □. The key point is that ◩ ⊕ ◪ = ■.

2.4 Assertion Language

We model the assertions of separation logic following Dockins *et al.* [11]. Our states σ are triples of a stack, heap, and barrier map ($\sigma = (s, h, b)$). Local variables live in stacks s (functions from variable names to values). In contrast, a heap h contains the locations shared between threads; heaps are partial functions from addresses to pairs of positive shares and values. We also equip our heaps with a distinguished location, called the *break*, that tracks the boundary between allocated and unallocated locations. The break lets us provide semantics for the x:= `new` e instruction in a natural way by setting x equal to the current break and then incrementing the break. Since threads share a common break, there is a backdoor communication channel; however the existence of this channel is a small price to pay for avoiding the necessity of a concurrent garbage collector. We ensure that the threads see the same break by equipping our break with ownership shares just as we equip normal memory locations with shares.

We denote the empty heap (which lacks ownership for both all memory locations and the distinguished break location) by h_0. Of note, our expressions e are evaluated only in the context of the stack; we write $s \vdash e \Downarrow v$ to mean that e evaluates to v in the context of the stack s. Finally, the barrier map b is a partial function from barrier numbers to pairs of barrier states (represented as natural numbers) and positive shares; we denote the empty barrier map by b_0.

An *assertion* is a function from states to truth values (`Prop` in Coq). As is common, we define the usual logical connectives via a straightforward embedding into the metalogic; for example, the object-level conjunction $P \wedge Q$ is defined as $\lambda\sigma.\ (P\sigma) \wedge (Q\sigma)$. We will adopt the convention of using the same symbol for both the object-level operators and the meta-level operators to avoid symbol bloat; it should be clear from the context which operator applies in a given situation. We provide all of the standard connectives ($\top, \bot, \wedge, \vee, \Rightarrow, \neg, \forall, \exists$).

We model the connectives of separation logic in the standard way[2]:

[2] Our Coq definition for `emp` is different but equivalent to the definition given here.

$$
\begin{array}{lcl}
\mathsf{emp} & = & \lambda(s,h,b).\ h = h_0 \ \wedge\ b = b_0 \\
P * Q & = & \lambda\sigma.\ \exists\sigma_1,\sigma_2.\ \sigma_1 \oplus \sigma_2 = \sigma \ \wedge\ P(\sigma_1) \ \wedge\ Q(\sigma_2) \\
e_1 \overset{\pi}{\mapsto} e_2 & = & \lambda(s,h,b).\ \exists a, v.\ (s \vdash e_1 \Downarrow \mathtt{ADDR}(a)) \ \wedge\ (s \vdash e_2 \Downarrow v) \ \wedge \\
 & & \quad b = b_0 \ \wedge\ h(a) = (v,\pi) \ \wedge\ \mathsf{dom}(h) = \{a\} \ \wedge\ \mathsf{break}(h) = \square \\
\mathsf{barrier}(bn,\pi,s) & = & \lambda(s,h,b).\ h = h_0 \ \wedge\ b(bn) = (s,\pi) \ \wedge\ \mathsf{dom}(b) = \{bn\}
\end{array}
$$

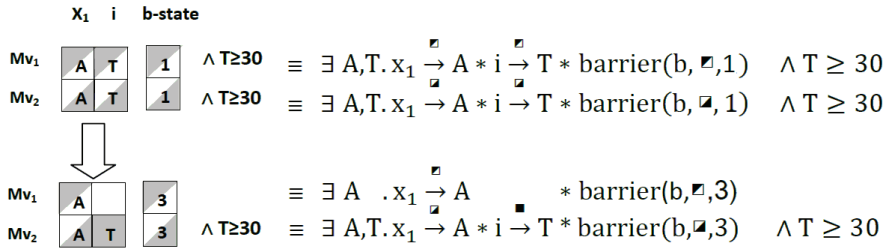
The fractional maps-to assertion, $e_1 \overset{\pi}{\mapsto} e_2$, means that the expression e_1 is pointing to an address a in memory; a is owned with positive share π, and contains the evaluated value v of e_2. The fractional maps-to assertion does not include any ownership of the break. The barrier assertion, $\mathsf{barrier}(bn,\pi,s)$, means that the barrier bn, owned with positive share π, is in state s.

We also lift program expressions into the logic: $e \Downarrow v$, which evaluates e with σ's stack (*i.e.*, $\lambda(s,h,b).\ h = h_0 \wedge b = b_0 \wedge s \vdash e \Downarrow v$); $[e]$, equivalent to $e \Downarrow \mathtt{TRUE}$; and $x = v$, equivalent to $\mathtt{V}(x) \Downarrow v$. These assertions have a "built-in" emp.

3 Example

We present a detailed example inspired by a video decompression algorithm. The code and a detailed-but-informal description of the barrier definition is given in Figure 1.[3] Two threads cooperate to repeatedly compute the elements of two size-two arrays `x` and `y`. In each iteration, each thread writes to a single cell of the "current" array, and reads from both cells of the "previous" array.

In Figure 1 we give a pictorial representation of the state machine associated with the barrier used in the code using the following specialized notation:



This notation is used to express the pre- and postconditions for a given barrier transition. Each row is a pictorial representation (values, barrier states, and shares) of a formula in separation logic as indicated above. The preconditions are on top (one per row) and the postconditions below. Each row is associated with a *move*; move 1 is a pair of the first precondition row and the first postcondition row, etc. A barrier that is waiting for n threads will have n moves; n can be fewer than the total number of threads. We do not require that a given thread always takes the same move each time it reaches a given barrier transition.

Note that only the permissions on the memory cells change during a transition; the contents (values) do not.[4] The exception to this is the special column

[3] In our Coq development we give the full formal description of the example barrier.

[4] We use the same quantified variable names before and after the transition because an outside observer can tell that the values are the same. A local verification can

0: $\{x_1 \overset{\blacksquare}{\longmapsto} 0 * x_2 \overset{\blacksquare}{\longmapsto} 0 * y_1 \overset{\blacksquare}{\longmapsto} 0 * y_2 \overset{\blacksquare}{\longmapsto} 0 * i \overset{\blacksquare}{\longmapsto} 0 * \mathsf{barrier}(bn, \blacksquare, 0)\}$

0': $\{x_1 \overset{\boxslash}{\longmapsto} 0 * x_2 \overset{\boxslash}{\longmapsto} 0 * y_1 \overset{\boxslash}{\longmapsto} 0 * y_2 \overset{\boxslash}{\longmapsto} 0 * i \overset{\boxslash}{\longmapsto} 0 * \mathsf{barrier}(bn, \boxslash, 0)\}$

$\{x_1 \overset{\boxslash}{\longmapsto} 0 * x_2 \overset{\boxslash}{\longmapsto} 0 * y_1 \overset{\boxslash}{\longmapsto} 0 * y_2 \overset{\boxslash}{\longmapsto} 0 * i \overset{\boxslash}{\longmapsto} 0 * \mathsf{barrier}(bn, \boxslash, 0)\}$

```
    ...                                          ...
1:  barrier b;                                   barrier b;        // b transitions 0→1
2:  n := 0;                                      m := 0;
3:  while n < 30 {                               while m < 30 {
4:      a1 := [x1];                                  a1 := [x1];
5:      a2 := [x2];                                  a2 := [x2];
6:      [y1] := (a1+2*a2);                           [y2] := (a1+3*a2);
7:      barrier b;                                   barrier b;    // b transitions 1→2
8:      a1 := [y1];                                  a1 := [y1];
9:      a2 := [y2];                                  a2 := [y2];
10:     [x1] := (a1+2*a2);                           [x2] := (a1+3*a2);
11:     n := (n+1);
12:     [i] := n;
13:     barrier b;                                   barrier b;    // b transitions 2→1
14:                                                  m := [i];
15: }                                            }
16: barrier b;                                   barrier b;        // b transitions 1→3
17: [i] := 0;
    ...                                          ...
```

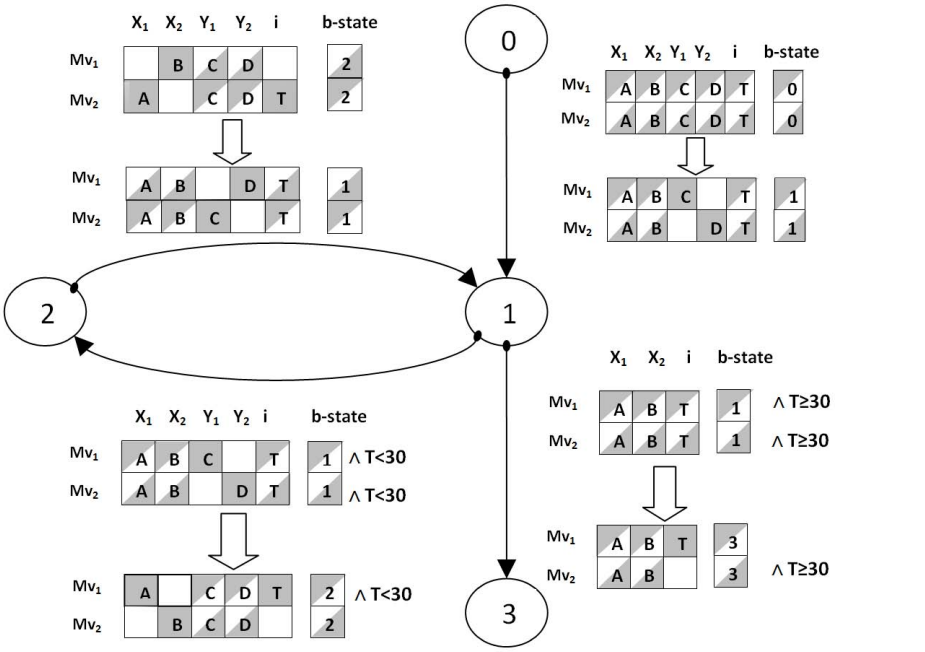


Fig. 1. Example: Code and Barrier Diagram

use ghost state to prove the equality; alternatively we could add the ability to move the quantifier to other parts of the diagram, *e.g.*, over an entire pre-post pair.

on the right side, which denotes the assertion associated with the barrier itself. As the barrier transitions, this value changes from the previous state to the next; we require that the sum of the preconditions includes the full share of the barrier assertion to guarantee that no thread has an out-of-date view of the barrier's state. Observe that all of the preconditions join together, and, except for the state of the barrier itself, are exactly equal to the join of the postconditions.

The initial state of the machine is given as an assertion in line 0. The machine starts with full ownership of the array cells x_1, x_2, y_1, and y_2, as well as an additional cell i, used as a condition variable. The barrier b is fully-owned and is in state 0. The initial state is then partitioned into two parts on line 0', with the left thread (A) and right thread (B) getting the shares ◩ and ◪, respectively.

Not shown (between lines 0' and 1) is thread-specific initialization code; perhaps both threads read both arrays and perform consistency checks. The real action starts with the barrier call on line 1, which ensures that this initialization code has completed. Thread A takes move 1 and thread B takes move 2. Afterwards, thread A has full ownership over y_1 and thread B has full ownership over y_2; the ownership of x_1, x_2, and i remains split between A and B. While the ownership of the barrier is unchanged, it is now in state 1.

We then enter the main loop on line 3. On lines 4–5, both threads read from the shared cells x_1 and x_2, and on line 6 both threads update their fully-owned cell. The barrier call on line 7 ensures that these updates have been completed before the threads continue. Since the value T at memory location i is less than 30, only the 1–2 transition is possible; the 1–3 transition requires T$\geq$ 30. Thread A takes move 1 and thread B takes move 2[5]; afterwards, both threads have partial shares of y_1 and y_2, thread A has the full share of x_1 and the condition cell i, and thread B has the full share of x_2; the barrier is in state 2.

Lines 8–10 are mirrors of lines 4–6. On lines 11–12, thread A updates the condition cell i. The barrier on line 13 ensures that the updates on lines 10 and 12 have completed before the threads continue; thread A takes move 2 while thread B takes move 1. Afterwards, the threads have the same permissions they had on entering the loop: A has full ownership of y_1, B has full ownership of y_2, and they share ownership of x_1, x_2, and i; the barrier is again in state 1.

On line 14, thread B reads from the condition variable i, and then the program loops back to line 3. After 30 iterations, the loop exits and control moves to the barrier on line 16. Observe that since the (shared) value T at memory location i is greater than or equal to 30, only the 1–3 transition is possible; the 1–2 transition requires T$<$ 30. Thread A takes move 1 while thread B takes move 2; afterwards, both threads are sharing ownership of x_1, x_2, y_1, and y_2 (since the transition from 1 to 3 does not mention y_1 and y_2 they are unchanged). Thread A has full permission over the condition variable i; the barrier is in state 3. Finally, on line 17, thread A updates i; the barrier on line 16 ensures that thread B's read of i on line 14 has already occurred.

[5] In this example a given thread always takes the same move for a given transition; however, this is not forced by the rules of our logic.

BarDef	≡	{ bd_bn : *Nat*	barrier id
(barrier definition)		bd_limit : *Nat*	# of threads
		bd_states : *list* BarStateDef}	state list
BarStateDef	≡	{bsd_bn : *Nat*	barrier id
(barrier state)		bsd_cs : *Nat*	state id
		bsd_directions : *list* BarMoveList	transition list
		bsd_limit : *Nat*}	# of threads
BarMoveList	≡	{bml_dest : *Nat*	next state
(transition)		bml_bn : *Nat*	barrier id
		bml_cs : *Nat*	current state
		bml_limit : *Nat*	# of threads
		bml_moves : *list* (assert × assert)}	pre/post pairs

Fig. 2. Barrier Definitions

4 Barrier Definitions and Consistency Requirements

We present the type of a barrier definition in Figure 2 in the form of a data structure. The definitions include numerous consistency requirements; in Coq these are maintained with dependent types. From the top down, a barrier definition (BarDef) consists of a barrier identifier (*i.e.*, barrier number), the number of threads the barrier is synchronizing, and a list of barrier state definitions. For programs that have more than one barrier, the individual barrier definitions will be collected into a list and barrier number j will be in list slot j.

A barrier state definition (BarStateDef) consists of a barrier number, the number of threads synchronized, a state id, and a transition list; such that:

1. the barrier number matches the barrier number in the containing BarDef
2. the limit matches the limit of the containing BarDef[6]
3. the state identifier j indicates that this BarStateDef is the j element of the containing BarDef's list of state definitions
4. the directions are *mutually exclusive*

The first three are unexciting; we will discuss mutual exclusion shortly.

A transition (BarMoveList) contains a barrier number (bn), number of threads synchronized, current state identifier (cs), next state identifier (ns), and list of precondition/postcondition pairs (the *move list*). We require that:

1. bn matches the barrier number in the containing BarStateDef
2. the limit matches the limit in the containing BarStateDef
3. cs matches the state identifier in the containing BarStateDef
4. the length of list of moves (bml_moves) is equal to the limit (bml_limit)
5. all of the pre/postconditions in the movelist ignore the stack, focusing only on the memory and barrier map. Since stacks are private to each thread (on a processor these would be registers), it does not make sense for them to be mentioned in the "public" pre/post conditions.

[6] A command to dynamically alter the number of threads a barrier managed might allow different states/transitions to wait for different numbers of threads.

6. all of the preconditions in the movelist are *precise*. Precision is a technical property involving the identifiability of states satisfying an assertion.[7]
7. each precondition P includes some positive share of the $\mathsf{barrier}$ assertion with bn and cs, *i.e.*, $\exists \pi.\ P \Rightarrow \top * \mathsf{barrier}(\mathsf{bn}, \pi, \mathsf{cs})$.
8. the sum of the preconditions must equal the sum of the postconditions, except for the state of the barrier; moreover, the sum of the preconditions must include the full share of the barrier (equation (2), repeated here):

$$\begin{aligned} \mathop{\Large *}_i\ Pre_i \quad &= \quad F * \mathsf{barrier}(\mathsf{bn}, \blacksquare, \mathsf{cs}) \\ \mathop{\Large *}_i\ Post_i \quad &= \quad F * \mathsf{barrier}(\mathsf{bn}, \blacksquare, \mathsf{ns}) \end{aligned}$$

Items 1–4 are simple bookkeeping; items 5–7 are similar to technical requirements required in other variants of concurrent separation logic [18, 14, 13]. As previously mentioned, the fundamental insight of this approach is property (8).

The function $\mathsf{lookup\_move}$ simplifies the lookup of a move in a BarDef:

$$\mathsf{lookup\_move}(bd, cs, dir, mv) = bd.\mathsf{bd\_states}[cs].\mathsf{bsd\_directions}[dir].\mathsf{bml\_moves}[mv]$$

Using this notation, we can express the important requirement that all directions in the barrier state cs of the barrier definition bd are mutually exclusive:

$$\begin{aligned} \forall dir_1, dir_2, mv_1, mv_2, pre_1, pre_2.\ dir_1 \neq dir_2 \ \Rightarrow \\ \mathsf{lookup\_move}(bd, cs, dir_1, mv_1) = (pre_1, \_) \ \Rightarrow \\ \mathsf{lookup\_move}(bd, cs, dir_2, mv_2) = (pre_2, \_) \ \Rightarrow \\ (\top * pre_1) \ \wedge \ (\top * pre_2) \ \equiv \ \bot \end{aligned}$$

In other words, it is *impossible* for any of the preconditions of more than one transition (of a given state) to be true at a time. The simplest way to understand this is to consider the 1–2 and 1–3 transitions in the example program. The 1–2 transition requires that the value in memory cell i be strictly less than 30; in contrast, the 1–3 transition requires that *the same cell* contains a value greater than or equal to 30. Plainly these are incompatible; but in fact the above property is stronger: *both* of the moves on the 1–2 transition, and *both* of the moves on the 1–3 transition include the incompatibility. Thus, if thread A takes transition 1–2, it knows for certain that thread B *cannot* take transition 1–3. This way we ensure that both threads always agree on the barrier's current state.

5 Hoare Logic

Our Hoare judgment has the form $\Gamma \vdash \{P\}\ c\ \{Q\}$, where Γ is a list of barrier definitions as given in §4, P and Q are assertions in separation logic, and c is a command. Our Hoare rules come in three groups: standard Hoare logic (Skip, If, Sequence, While, Assignment, Consequence); standard separation logic (Frame, Store, Load, New, Free); and the barrier rule. We give groups two and three in Figure 3; group one is standard and elided. We note four points for group two.

[7] Precision may not be required; another property (tentatively christened "token") that might serve would be if, for any precondition P, $P * P \equiv \bot$. Note that precision in conjunction with item (6) implies P is a token.

$$\frac{\Gamma \vdash \{P\}\ c\ \{Q\}\quad \mathsf{closed}(F, c)}{\Gamma \vdash \{F * P\}\ c\ \{F * Q\}}\ \text{Frame} \qquad \frac{}{\Gamma \vdash \{e_1 \overset{\blacksquare}{\mapsto} \_\}\ [e_1] := e_2\ \{e_1 \overset{\blacksquare}{\mapsto} e_2\}}\ \text{Store}$$

$$\frac{}{\Gamma \vdash \{e_1 \overset{\pi}{\mapsto} e_2 * e_1 \Downarrow v_1 * e_2 \Downarrow v_2\}\ x := [e_1]\ \{\mathtt{C}(v_1) \overset{\pi}{\mapsto} \mathtt{C}(v_2) * x = v_2\}}\ \text{Load}$$

$$\frac{}{\Gamma \vdash \{e \Downarrow v\}\ x := \mathtt{new}\ e\ \{\mathtt{V}(x) \overset{\blacksquare}{\mapsto} \mathtt{C}(v)\}}\ \text{New} \qquad \frac{}{\Gamma \vdash \{e_1 \overset{\blacksquare}{\mapsto} e_2\}\ \mathtt{free}\ e_1\ \{\mathsf{emp}\}}\ \text{Free}$$

$$\boxed{\frac{\Gamma[bn] = bd \qquad \mathsf{lookup\_move}(bd, cs, dir, mv) = (P, Q)}{\Gamma \vdash \{P\}\ \mathbf{barrier\ bn}\ \{Q\}}\ \text{Barrier}}$$

Fig. 3. Hoare rules (not pictured: Skip, If, Sequence, While, Assign, and Consequence)

First, as explained in §2.4, the assertions $e \Downarrow v$, $[e]$ and $x = v$ are bundled with an assertion that the heap and barrier map are empty(*i.e.*, $e \Downarrow v \ \Rightarrow\ \mathsf{emp}$); thus, we use the separating conjunction when employing them. Second, the rules are in "side-condition-free form". Thus, instead of presenting the load rule as $\Gamma \vdash \{e_1 \overset{\pi}{\mapsto} e_2\}\ x := [e_1]\ \{x = e_2 * e_1 \overset{\pi}{\mapsto} e_2\}$, which is aesthetically attractive but untrue in the pesky case when e_2 depends on x (*e.g.*, $x := [x]$), we use a form that is less visually pleasing but does not require side conditions.[8] It is straightforward to restore rules with side conditions via the Consequence rule. Third, our Store and Free rules require the full share of location e_1; in contrast, our Load rule only requires some positive share; this is consistent with our use of fractional permissions as explained in §2.3. Fourth, memory allocation and deallocation are more complicated in concurrent settings than in sequential settings, and so the New and Free rules cause nontrivial complications in the semantic model.

The Hoare rule for barriers is so simple that at first glance it may be hard to understand. The variables for the current state *cs*, direction *dir*, and move *mv* appear to be free in the lookup_move! However, things are not quite as unconstrained as they initially appear. Recall from §4 that one of the consistency requirements for the precondition P is that P implies an assertion about the barrier itself: $P \Rightarrow Q * \mathsf{barrier}(bn, \pi, cs)$; thus at a given program point we can only use directions and moves from the current state. Similarly, recall from §4 that since the directions are mutually exclusive, *dir* is uniquely determined.

This leaves the question of the uniqueness of *mv*. If a thread only satisfies a single precondition, then the move *mv* is uniquely determined. Unfortunately, it is simple to construct programs in which a thread enters a barrier while satisfying the preconditions of multiple moves. What saves us is that we are developing a logic of partial correctness. Since preconditions to moves must be precise and nonempty (*i.e.*, token), only one thread is able to satisfy a given precondition at a time. The pigeonhole principle guarantees that if a thread holds multiple preconditions then some other thread will not be able to enter the barrier; in this case, the barrier call will never return and we can guarantee any postcondition.

[8] Recall from §2: $\mathtt{V}(x)$ and $\mathtt{C}(v)$ are expression constructors for locals and constants. In addition, $\mathsf{closed}(F, c)$ means that F does not depend on locals modified by c.

We now apply the Barrier rule to the barrier calls in line 13 from our example program; the lookup_moves are direct from the barrier state diagram:

$$\text{Thread A}\ \begin{cases} \mathsf{lookup\_move}(b,2,1,2) = (P,Q) \\ P = y_1 \overset{\blacksquarefill}{\longmapsto} v_{y1} * y_2 \overset{\blacksquarefill}{\longmapsto} v_{y2} * x_1 \overset{\blacksquare}{\longmapsto} v_{x1} * i \overset{\blacksquare}{\longmapsto} v_i * \mathsf{barrier}(bn, \blacksquarefill, 2) \\ Q = y_1 \overset{\blacksquare}{\longmapsto} v_{y1} * x_1 \overset{\blacksquarefill}{\longmapsto} v_{x1} * x_2 \overset{\blacksquarefill}{\longmapsto} v_{x2} * i \overset{\blacksquarefill}{\longmapsto} v_i * \mathsf{barrier}(bn, \blacksquarefill, 1) \\ \hline \Gamma \vdash \{P\}\ \mathbf{barrier\ b}\ \{Q\} \end{cases}$$

$$\text{Thread B}\ \begin{cases} \mathsf{lookup\_move}(b,2,1,1) = (P,Q) \\ P = y_1 \overset{\blacksquarehalf}{\longmapsto} v_{y1} * y_2 \overset{\blacksquarehalf}{\longmapsto} v_{y2} * x_2 \overset{\blacksquare}{\longmapsto} v_{x2} * \mathsf{barrier}(bn, \blacksquarehalf, 2)\} \\ Q = y_2 \overset{\blacksquare}{\longmapsto} v_{y2} * x_1 \overset{\blacksquarehalf}{\longmapsto} v_{x1} * x_2 \overset{\blacksquarehalf}{\longmapsto} v_{x2} * i \overset{\blacksquarehalf}{\longmapsto} v_i * \mathsf{barrier}(bn, \blacksquarehalf, 1)\} \\ \hline \Gamma \vdash \{P\}\ \mathbf{barrier\ b}\ \{Q\} \end{cases}$$

Note that in this line of the example program, the frame is emp in both threads.

Not shown in Figure 3 is a parallel composition rule. As in [14], each thread is verified independently using the Hoare rules given; a top-level safety theorem proves that the entire concurrent machine behaves as expected.

6 Semantic Models

Our operational semantics is divided into three parts: purely sequential, which executes all of the instructions except for barrier in a thread-local manner; concurrent, which manages thread scheduling and handles the barrier instruction; and oracular, which provides a pseudosequential view of the concurrent machine to enable simple proofs of the sequential Hoare rules. Our setup follows Hobor *et al.* very closely and we refer readers there for more detail [15, 14].

Purely sequential semantics. The purely sequential semantics executes the instructions `skip`, $x := e$, $x := [e]$, $[e_1] := e_2$, x:= `new` e, `free` e, c_1; c_2, `if` e `then` c_1 `else` c_2, and `while` e $\{c\}$. The form of the sequential step judgment is $(\sigma, c) \mapsto (\sigma', c')$. Here σ is a state (triple of stack, heap, barrier map), just as in §2.4 and c is a command of our language. The semantics of the sequential instructions is standard; the only "tricky" part is that the machine gets stuck if one tries to write to a location for which one does not have full permission or read from a location for which one has no permission; *e.g.*, here is the store rule:

$$\frac{\begin{array}{ccc} s \vdash e_1 \Downarrow \mathtt{C}(\mathtt{ADDR}(n)) & & s \vdash e_2 \Downarrow v \\ n < \mathsf{break}(h) & h(n) = (\blacksquare, v') & h' = [n \mapsto (\blacksquare, v)]h \end{array}}{\big((s,h,b),\ [e_1] := e_2;\ c\big)\ \mapsto\ \big((s,h',b),c\big)}\ \mathsf{sstep-store}$$

The test that $n < \mathsf{break}(h)$ ensures that the address for the store is "in bounds"—that is, less than the current value of the break between allocated and unallocated memory; since we are updating the memory we require that the permission associated with the location n full ($\blacksquare$). We say that this step relation is *unerased* since these bounds and permission checks are virtual rather than on-chip.

We define the other cases of the step relation in a similar way. Observe that if we were in a sequential setting the proof of the Hoare store rule would be straightforward; this is likewise the case for the other cases of the sequential step relation and their associated Hoare rules. If the sequential step relation reaches a barrier call **barrier** bn then it simply gets stuck.

Concurrent semantics. We define the notion of a *concurrent state* in Figure 4. A concurrent state contains a scheduler Ω (modeled as a list of natural numbers), a distinguished heap called the *allocation pool*, a list of *thread*s, and a *barrier pool*[9]. The allocation pool is the owner of all of the unallocated memory cells (plus the ownership of the break between allocated and unallocated cells); before we run a thread we transfer the allocation pool into the local heap owned by the thread so that `new` can transfer a cell from this pool into the local heap of a thread when required. When we suspend the thread we remove (what is left of) the allocation pool from its heap so that we can transfer it to the next thread.

A thread contains a (sequential) state (stack, heap, and barrier map) and a *concurrent control*, which is either $\mathsf{Running}(c)$, meaning the thread is available to run command c, or $\mathsf{Waiting}(bn, dir, mv, c)$, meaning that the thread is currently waiting on barrier bn to make move mv in direction dir; after the barrier call completes the thread will resume running with command c.

The barrier pool (Barpool) contains a list of *dynamic barrier status*es (DBSes) as well as a state which is the join of all of the states inside the DBSes. Each DBS consists of a barrier number (which must be its index into the array of its containing Barpool), a barrier definition (from §4), and a *waitpool* (WP). A waitpool consists of a direction option (`None` before the first barrier call in a given state; thereafter the unique direction for the next state), a limit (the number of threads synchronized by the barrier, and comes from the barrier definition in the enclosing DBS), a *slot* list, and a state (which is the join of all of the states in the slot list). A slot is a heap and barrier map (the stack is unneeded since barrier pre/postconditions ignore it) as well as a thread id (whence the heap and barrier map came as a precondition, and to which the postcondition will return).

The concurrent step relation is $(\Omega, ap, thds, bp) \rightsquigarrow (\Omega', ap', thds', bp')$, where Ω, ap, $thds$, and bp are the scheduler, allocation pool, thread list, and barrier pool respectively. The concurrent step relation has only four cases; the following case CStep-Seq is used to run all of the sequential commands:

$$\frac{thds[i] = (s, h, b, \mathsf{Running}(c)) \qquad h \oplus ap = h' \qquad \big((s, h', b), c\big) \mapsto \big((s', h'', b), c'\big) \\ h''' \oplus ap' = h'' \qquad \mathsf{isAllocPool}(ap') \qquad thds' = [i \mapsto (s', h''', b, \mathsf{Running}(c'))]thds}{(i :: \Omega, thds, ap, bp) \rightsquigarrow (i :: \Omega, thds', ap', bp)} \text{ CStep-Seq}$$

That is, we look up the thread whose thread id is at the head of the scheduler, join in the allocation pool, and run the sequential step relation. If the command c is a barrier call then the sequential relation will not be able to run and so

[9] There is also a series of consistency requirements such as the fact that all of the heaps in the threads and barrier pool join together with the allocation pool into one consistent heap; in the mechanization this is carried around via a dependent type as a fifth component of the concurrent state. We elide this proof from the presentation.

Cstate	$\equiv$	{ cs_sched : list $\mathbb{N}$	schedule
		cs_allocpool : heap	alloc pool
		cs_thds : list Thread	thread pool
		cs_barpool : Barpool}	barrier pool
Thread	$\equiv$	{ th_stk : stack	
		th_hp : heap	
		th_bs : BarrierMap	local view of barrier states
		th_ctl : conc_ctl}	running or waiting
conc_ctl	$\equiv$	| Running(c)	executing code c
		| Waiting(bn, dir, mv, c)	waiting on bn
Barpool	$\equiv$	{ bp_bars : list DyBarStatus	dynamic barrier status
		bp_st : stack $\times$ heap $\times$ BarrierMap}	current state
DyBarStatus	$\equiv$	{ dbs_bn : $\mathbb{N}$	barrier id
		dbs_wp : Waitpool	waiting thread pool
		dbs_bd : BarDef}	
Waitpool	$\equiv$	{ wp_dir : $\mathbb{N}$ *option*	direction id
		wp_slots : list slot *option*	taken slots
		wp_limit : $\mathbb{N}$	
		wp_st : stack $\times$ heap $\times$ BarrierMap}	current state
slot	$\equiv$	(thread_id $\times$ heap $\times$ BarrierMap)	waiting slot

Fig. 4. Concurrent state

the CStep-Seq relation will not hold; otherwise the sequential step relation will be able to handle any command. After we have taken a sequential step, we subtract out the (possibly diminished) allocation pool, and reinsert the modified sequential state into the thread list. Since we quantify over all schedulers and our language does not have input/output, it is sufficient to utilize a non-preemptive scheduler; for further justification on the use of such schedulers see [14].

The second case of the concurrent step relation handles the case when a thread has reached the last instruction, which must be a `skip`:

$$\frac{thds[i] = \mathsf{Running}(\texttt{skip})}{(i :: \Omega, thds, ap, bp) \rightsquigarrow (\Omega, thds, ap, bp)}\ \text{CStep-Exit}$$

When we reach the end of a thread we simply context switch to the next thread.

The interesting cases occur when the instruction for the running thread is a barrier call; here the CStep-Seq rule does not apply. The concurrent semantics handles the barrier call directly via the last two cases of the step relation; before presenting these cases we will first give a technical definition called fill_barrier_slot:

$$\frac{\begin{array}{c} thds[i] = \mathsf{Thread}(stk,\ hp,\ bs,\ (\mathsf{Running}\ (barrier\ bn\ ;\ c))) \\ \mathsf{lookup\_move}(bp.bp\_bars[bn],\ dir,\ mv) = (pre, post) \\ hp' \oplus hp'' = hp \qquad bs' \oplus bs'' = bs \qquad pre(stk, hp', bs') \\ \mathsf{bp\_inc\_waitpool}\ (bp,\ bn,\ dir,\ mv,\ (i,\ (hp',\ bs'))) = bp' \\ thds' = [i \rightarrow (\mathsf{Thread}(\ stk,\ hp'',\ bs'',\ (\mathsf{Waiting}\ (bn,\ dir,\ mv,\ c))))]\ thds \end{array}}{\mathsf{fill\_barrier\_slot}\ \ (thds,\ bp,\ bn,\ i) = (thds',\ bp')}$$

The predicate fill_barrier_slot gives the details of removing the (sub)state satisfying the precondition of the barrier from the thread's state, inserting it into the barrier pool, and suspending the calling thread. The predicate bp_inc_waitpool does the insertion into the barrier pool; the details of manipulating the data structure are straightforward but lengthy to formalize[10].

We are now ready to give the first case for the barrier, used when a thread executes a barrier but is not the last thread to do so:

$$\frac{\begin{array}{c}\mathsf{fill\_barrier\_slot}\ \ (thds,\ bp,\ bn,\ i) = (thds',\ bp') \\ \neg\ \mathsf{bp\_ready}\ \ (bp',\ bn)\end{array}}{((i :: \Omega), ap, thds, bp) \rightsquigarrow (\Omega, ap, thds', bp')}\ \text{CStep-Suspend}$$

After using fill_barrier_slot, CStep-Suspend checks to see if the barrier is full by counting the number of slots that have been filled in the appropriate wait pool by using the bp_ready predicate, and then context switches.

If the barrier is ready then instead of using the CStep-Suspend case of the concurrent step relation, we must use the CStep-Release case:

$$\frac{\begin{array}{c}\mathsf{fill\_barrier\_slot}\ \ (thds,\ bp,\ bn,\ i) = (thds',\ bp') \\ \mathsf{bp\_ready}\ \ (bp',\ bn) \\ \mathsf{bp\_transition}\ \ (bp',\ bn,\ out) = bp'' \\ \mathsf{transition\_threads}\ \ (out,\ thds') = thds''\end{array}}{((i :: \Omega), ap, thds, bp) \rightsquigarrow (\Omega, ap, thds'', bp'')}\ \text{CStep-Release}$$

The first requirement of CStep-Release is exactly the same as CStep-Suspend: we suspend the thread and transfer the appropriate resources to the barrier pool. However, now all of the threads have arrived at the barrier and so it is ready. We use the bp_transition predicate to go through the barrier's slots in the waitpool, combine the associated heaps and barrier maps, redivide these resources according to the barrier postconditions, and remove the associated resources from the barrier pool into a list of slots called out. Finally, the states in out are combined with the suspended threads, which are simultaneously resumed by the transition_threads predicate. The formal definitions of the bp_transition and transition_threads predicates are extremely complex and very tedious and we refer interested readers to the mechanization.

Oracle semantics. Following Hobor *et al.* [15, 14], we define a third *oracular semantics*: $(\sigma, o, c) \mapsto (\sigma', o', c')$. Here the sequential state σ and command c are exactly the same as in the purely sequential step. The new parameter o is an oracle, a kind of box containing "the rest" of the concurrent machine—that is, o contains a scheduler, a list of other threads, and a barrier pool.

The oracle semantics behaves exactly the same way as the purely sequential semantics on all of the instructions except for the barrier call, with the oracle o being passed through unchanged. That is to say:

$$\frac{(\sigma, c)\ \mapsto\ (\sigma', c')}{(\sigma, o, c)\ \mapsto\ (\sigma', o, c')}\ \text{os-seq}$$

[10] In Coq things are trickier since we track some technical side conditions via dependent types so this relation also ensures that these side conditions remain satisfied.

When the oracle semantics reaches a barrier instruction, it consults the oracle o to determine the state of the machine after the barrier:

$$\frac{\mathsf{consult}(h, b, o) = (h', b', o')}{\big((s, h, b), o, \mathbf{barrier}\ \mathsf{bn};\ c\big)\ \mapsto\ \big((s, h', b'), o', c\big)}\ \text{os-consult}$$

The formal definition of the consult relation is detailed in [15, 14] but the idea is simple. To consult the oracle, one unpacks the concurrent machine and runs (classically) all of the other threads until control returns to the original thread; consult then returns the current h' and b' (that resulted from the barrier call) and repackages the concurrent machine into the new oracle o'. The final case of the oracle semantics occurs when the concurrent machine never returns control (because it got stuck or due to sheer perversion of the scheduler):

$$\frac{\nexists r.\ \mathsf{consult}(h, b, o) = r \qquad (\textit{i.e.}, \mathsf{consult}\ \text{diverges})}{\big((s, h, b), o, \mathbf{barrier}\ \mathsf{bn};\ c\big)\ \mapsto\ \big((s, h, b), o, \mathbf{barrier}\ \mathsf{bn};\ c\big)}\ \text{os-diverge}$$

When control will never return, it does not matter what this thread does as long as it does not get stuck; accordingly we enter an (infinite) loop.

Soundness proof outline. Our soundness argument falls into several parts. We define our Hoare tuple in terms of our oracle semantics using a definition by Appel and Blazy [2]; this definition was designed for a sequential language and we believe that other standard sequential definitions for Hoare tuples would work as well[11]. We then prove (in Coq) all of the Hoare rules for the sequential instructions; since the os-seq case of the oracle semantics provides a straight lift into the purely sequential semantics this is straightforward[12].

Next, we prove (in Coq) the soundness for the barrier rule. This turns out to be much more complicated than a proof of the soundness of (non-first-class) locks and took the bulk of the effort. There are two points of particular difficulty: first, the excruciatingly painful accounting associated with tracking resources during the barrier call as they move from a source thread (as a precondition), into the barrier pool, and redistribution to the target thread(s) as postcondition(s). The second difficulty is proving that a thread that enters a barrier while holding more than one precondition will never wake up; the analogy is a door with n keys distributed among n owners; if an owner has a second key in his pocket when he enters then one of the remaining owners will not be able to get in.

After proving the Hoare rules from Figure 3 sound with respect to the oracle semantics, the remaining task is to connect the oracle semantics to the concurrent semantics—that is, *oracle soundness*. Oracle soundness says that if each of the threads on a machine are safe with respect to the oracle semantics, then the entire concurrent machine combining the threads together is safe. The (very rough)

[11] We change Appel and Blazy's definition so that our Hoare tuple guarantees that the allocation pool is available for verifying the Hoare rule for x:= `new` e.

[12] The Hoare rule for loops (While) is only proved on paper. The loop rule is known to be painful to mechanize and so the mechanization was skipped due to time constraints. It has been proved in Coq for similar (indeed, more complicated from a sequential control-flow perspective) settings in previous work [2, 15].

analogy to this result in Brookes' semantics is the parallel decomposition lemma. Here we use a progress/preservation style proof closely following that given in [14, *pp.*242–255]; the proof was straightforward and quite short to mechanize. A technical advance over previous work is that the progress/preservation proofs do not require that the concurrent semantics be deterministic. In fact, allowing the semantics to be nondeterministic simplified the proofs significantly.

A direct consequence of oracle soundness is that if each thread is verified with the Hoare rules, and is loaded onto a single concurrent machine, then if the machine does not get stuck and if it halts then all of the postconditions hold.

Erasure. One can justly observe that our concurrent semantics is not especially realistic; *e.g.*, we: explicitly track resource ownership permissions (*i.e.*, our semantics is *unerased*); have an unrealistic memory allocator/deallocator and scheduler; ignore issues of byte-addressable memory; do not store code in the heap; and so forth. We believe that we could connect our semantics to a more realistic semantics that could handle each of these issues, but most of them are orthogonal to barriers. For brevity we will comment only on erasing the resource accounting since it forms the heart of our soundness result.

We have defined, in Coq, an *erased* sequential and concurrent semantics. An erased memory is simply a pair of a break address and a total function from addresses to values. The run-time state of an erased barrier is simply a pair of naturals: the first tracking the number of threads currently waiting on the barrier, and the second giving the final number of threads the barrier is waiting for. We define a series of erase functions that take an unerased type (memory/barrier status/thread/etc.) to an erased one by "forgetting" all permission information. The sequential erased semantics is quite similar to the unerased one, with the exception that we do not check if we have read/write permission before executing a load/store. The concurrent erased semantics is much simpler than the complicated accounting-enabled semantics explained above since all that is needed to handle the barrier is incrementing/resetting a counter, plus some modest management of the thread list to suspend/resume threads. Critically, our erased semantics is a computable function, enabling program evaluation. Finally, we have proved that our unerased semantics is a conservative approximation to our erased one: that is, if our unerased concurrent machine can take a step from some state Σ to Σ', then our erased machine takes a step from $\mathsf{erase}(\Sigma)$ to $\mathsf{erase}(\Sigma')$.

7 Coq Development

We detail our Coq development in Figure 5. We use the Mechanized Semantic Library [1] for the definitions of share models, separation algebras, and various utility lemmas/tactics. In addition to the standard Coq axioms, we use dependent and propositional extensionality and the law of excluded middle.

Over 7,000 lines of the development is devoted to proving the soundness of the Hoare rule for barriers, largely in the files `SLB_BarDefs.v`, `SLB_CLang.v`, `SLB_Sem.v`, `SLB_OSem.v`, `SLB_HRules.v`, and a small part of `SLB_HRulesSound.v`. The rest of the concurrent semantics, the oracle semantics, and the soundness of the oracle semantics (∼the parallel decomposition lemma) require approximately

File	LOC	Time	Description
SLB_Base	1,182	2s	Utility lemmas (largely list facts)
SLB_Lang	1,240	11s	States, program syntax, assertion model
SLB_BarDefs	265	2s	Barrier definitions
SLB_CLang	3,230	1m7s	Dynamic concurrent state
SLB_SSem	415	17s	Sequential semantics
SLB_Sem	784	33s	Concurrent semantics
SLB_ESSem	230	5s	Erased semantics
SLB_ESEquiv	650	30s	Erasure proofs
SLB_OSem	1,942	2m10s	Oracular semantics
SLB_HRules	170	2s	Definition of Hoare tuples
SLB_OSound	426	30s	Soundness of oracle semantics
SLB_HRulesSound	1,664	1m14s	Soundness proofs for Hoare rules
SLB_Ex	2,700	48s	Example of a barrier definition
Total	14,898	7m34s	

Fig. 5. Proof structure, size and compilation times (2.66GHz, 8GB)

1,000 lines, largely in the files `SLB_Sem.v`, `SLB_HRules.v`, and `SLB_OSound`. The erased semantics requires 230 lines in the file `SLB_ESSem.v`, while the associated equivalence proofs require 650 lines in the file `SLB_ESEquiv.v`.

The sequential semantics and proofs for the associated Hoare rules require approximately 2,000 lines drawn from the files `SLB_Lang.v`, `SLB_SSem.v`, `SLB_HRules.v`, and `SLB_HRulesSound.v`. We estimate that the proof of the loop rule would require a further 2,000-3,000 lines. The model of our assertions and the program syntax are both in `SLB_Lang.v`. Utility lemmas/tactics (`SLB_Base.v`) and the example barrier (`SLB_Ex.v`) complete the development.

8 Limitations and Future Work

We have two obvious directions for future work. First, we can extend the logic by making the barriers first-class (i.e., dynamic barrier creation/destruction). In the present work we thought we could simplify the proofs by having statically declared barriers in the style of O'Hearn [18]. This turned out to be somewhat of a mistake: since we were forced to track the barrier states (and partial shares) explicitly in the Hoare logic, we estimate that 90% of the work required to make the barriers first-class has already been done in the present work; moreover, a further 8% (the intrinsic contravariant circularity) would be easy to handle via indirection theory [16]. With perfect foresight (or if it were nontrivial to restart a large mechanized proof), we would have certainly made the barriers first-class.

Second, we do not address the tricky problem of program analysis. One place where we believe that automatic program verification could be easily applied is in verifying that barrier definitions meet the various soundness requirements. We would also like to investigate verifying program text containing barrier calls; one place to begin is constructing a verifier for programs that use OpenMP [10].

9 Related Work

Calcagno *et al.* proposed separation algebras as models of separation logic [9]; fractional permissions were discussed by Bornat *et al.* [5]. In our work we use the share model and separation algebra development of Dockins *et al.* [11, 1].

O'Hearn's concurrent separation logic focused on programs that used critical regions [18, 6]; subsequent work by Hobor *et al.* and Gotsman *et al.* added first-class locks and threads [15, 13, 14]. Our basic soundness techniques (unerased semantics tracks resource accounting; oracle semantics isolates sequential and concurrent reasoning from each other; etc.) follow Hobor *et al.* Recently both Villard *et al.* and Bell *et al.* extended concurrent separation logic to channels [3, 19]. The work on channels is similar to ours in that both Bell and Villard track additional dynamic state in the logic and soundness proof. Bell tracks communication histories while Villard tracks the state of a finite state automata associated with each communication channel. Of all of the previous soundness results, only Hobor *et al.* had a machine-checked soundness proof; it was incomplete.

An interesting question is whether is it possible to reason about barriers in a setting with locks or channels. The question has both an operational and a logical flavor. Speaking operationally, in a practical sense the answer is no: for performance reasons barriers are not implemented with channels or locks. If we ignore performance, however, it **is** possible to implement barriers with channels or locks[13]. The logical part of the question then becomes, are the program logics defined by O'Hearn, Hobor, Gotsman, Villard, or Bell (including their coauthors) strong enough to reason about the (implementation of) barriers in the style of the logic we have presented? As far as we can tell each previous solution is missing at least one required feature, so in a strict sense, the answer here is again no.

For illustration we examine what seems to be the closest solution to ours: the copyless message passing channels of Villard *et al.* Operationally speaking, the best way to implement barriers seems to be by adding a central authority that maintains a channel with each thread using a barrier. When a thread hits a barrier, it sends "waiting" to the central authority, and then waits until it receives "proceed". In turn, the central authority waits for a "waiting" message from each thread, and then sends each of them a "proceed" message. Fortunately Villard allows the central authority to wait on multiple channels simultaneously.

The question then becomes a logical one. Although it should not pose any fundamental difficulty, their logic would first need to enhanced with fractional permissions; in fact we believe that Villard's Heap-Hop tool already uses the same fractional permission model (by Dockins *et al.*) that we do[14]. Since Villard uses automata to track state, we think it probable, but not certain, that our barrier state machines can be encoded as a series of his channel state machines.

There are some problems to solve. Villard requires certain side conditions on his channels; we require other kinds of side conditions on our barriers; these conditions do not seem fully compatible[15]. Assuming that we can weaken/strengthen conditions appropriately, we reach a second problem with the side conditions: some of our side conditions (*e.g.*, mutual exclusion) are restrictions on the shape

[13] Indeed, it is possible to implement channels and locks in terms of each other.

[14] To be precise, Heap-Hop uses the code extracted from the fractional permission Coq proof development by Dockins *et al.*

[15] For example, Villard requires determinacy whereas we do not; he would also require that the postconditions of barriers be precise whereas we do not; etc.

of the entire diagram; in Villard's setting the barrier state diagram has been partitioned into numerous separate channel state machines. Verifying our side conditions seems to require verification of the relationships that these channel state machines have to each other; the exact process is unclear.

Once the matter of side conditions is settled, there remains the issue of verifying the individual threads and the central authority. Villard's logic seems to have all that is required for the individual threads; the question is how difficult it would be to verify the central authority. Here we are less sure but suspect that with enough ghost state/instructions it can be done.

There remains a question as to whether it is a good idea to reason about barriers via channels (or locks). We suspect that it is not a good idea, even ignoring the fact that actual implementations of barriers do not use channels. The main problem seems to be a loss of intuition: by distributing the barrier state machine across numerous channel state machines and the inclusion of necessary ghost state, it becomes much harder to see what is going on. We believe that one of the major contributions of our work is that our barrier rule is extremely simple; with a quick reference to the barrier state diagram it is easy to determine what is going on. There is a secondary problem: we believe that our barrier rule will look and behave essentially the same way in a setting with first-class barriers in which it is possible to define functions that are polymorphic over the barrier diagram; even assuming a channel logic enriched in a similar way, the verification of a polymorphic central authority seems potentially formidable.

Finally, work on concurrent program analysis is in the early stages; Gotsman *et al.*, Calcagno *et al.*, and Villard *et al.* give techniques that cover some use cases involving locks and channels but much remains to be done [12, 8, 20].

Connection to an upcoming result by Jacobs and Piessens. We recently learned that Jacobs and Piessens have an impressive upcoming result on modular fine-grained concurrency [17]. Jacobs was able to reason about our example program using his VeriFast tool by designing an implementation of barriers using locks and reducing our barrier diagram to a large disjunction for a resource invariant. However, there are some costs. First, VeriFast requires the user to add annotations, such as function pre- and postconditions, loop invariants, folds/unfolds, etc. In the case of our 30-line example program, more than 600 lines of annotation were required, not including the code/annotiations for the barrier implementation itself; in contrast, using our logic, verifying the example program is extremely simple. Second, it was harder to gain insight into the program from the disjunction-form of the invariant; in contrast we find our barrier diagrams straightforward. Finally, it is unclear to us whether the reduction is always possible or whether it was only enabled by the relative simplicity of our example program. That said, Jacobs and Piessens have the only logic proven to be able to reason about barriers as derived from a more general mechanism.

10 Conclusion

We have designed and proved sound a program logic for Pthreads-style barriers. Our development includes a formal design for barrier definitions and a series

of soundness conditions to verify that a particular barrier can be used safely. Our Hoare rules can verify threads independently, enabling a thread-modular approach. Our soundness proof defines an operational semantics that explicitly tracks permission accounting during barrier calls and is machine-checked in Coq.

Acknowledgements. We thank Christian Bienia for showcasing numerous example programs containing barriers, Christopher Chak for help on an early version of this work, Jules Villard for useful comments in general and in particular on the relation of our logic to the logic of his Heap-Hop tool, and Bart Jacobs for discovering how to verify our example program in his VeriFast tool.

References

1. Andrew Appel, Robert Dockins, and Aquinas Hobor. Mechanized Semantic Library. Available at http://msl.cs.princeton.edu, 2009–2010.
2. Andrew W. Appel and Sandrine Blazy. Separation logic for small-step C minor. In *TPHOLs*, pages 5–21, 2007.
3. Christian J. Bell, Andrew W. Appel, and David Walker. Concurrent separation logic for pipelined parallelization. In *SAS*, 2010.
4. Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Department of Computer Science, Princeton, NJ, December 2010.
5. Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.
6. Stephen D. Brookes. A semantics for concurrent separation logic. In *CONCUR*, pages 16–34, 2004.
7. David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.
8. Cristiano Calcagno, Dino Distefano, and Vikor Vafeiadis. Bi-abductive resource invariant synthesis. In *APLAS*, 2009.
9. Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Symposium on Logic in Computer Science*, 2007.
10. Rohit Chandra, Ramesh Menon, Leonardo Dagum, Dave Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.
11. Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, 2009.
12. Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural Shape Analysis with Separated Heap Abstractions. In *SAS*, pages 240–260, 2006.
13. Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, pages 19–37, 2007.
14. Aquinas Hobor. Oracle semantics. Technical Report TR-836-08, Princeton, 2008.
15. Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, pages 353–367, 2008.
16. Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In *POPL '10*, pages 171–185, 2010.
17. Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL '11*, page To appear, 2011.
18. Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1):271–307, May 2007.
19. Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In *APLAS*, pages 194–209, 2009.
20. Jules Villard, Étienne Lozes, and Cristiano Calcagno. Tracking heaps that hop with heap-hop. In *TACAS*, pages 275–279, 2010.