Oracle Semantics for Concurrent Separation Logic (Extended Version)

Aquinas Hobor[1]⋆ Andrew W. Appel[1]⋆ Francesco Zappa Nardelli[2] ⋆⋆

[1] Princeton University [2] INRIA

Abstract. We define (with machine-checked proofs in Coq) a modular operational semantics for Concurrent C minor—a language with shared memory, spawnable threads, and first-class locks. By *modular* we mean that one can reason about sequential control and data-flow knowing almost nothing about concurrency, and one can reason about concurrency knowing almost nothing about sequential control and data-flow constructs. We present a Concurrent Separation Logic with first-class locks and threads, and prove its soundness with respect to the operational semantics. Using our modularity principle, we proved the sequential C.S.L. rules (those inherited from sequential Separation Logic) simply by adapting Appel & Blazy's machine-checked soundness proofs. Our Concurrent C minor operational semantics is designed to connect to Leroy's optimizing (sequential) C minor compiler; we propose our modular semantics as a way to adapt Leroy's compiler-correctness proofs to the concurrent setting. Thus we will obtain end-to-end proofs: the properties you prove in Concurrent Separation Logic will be true of the program that actually executes on the machine.

1 Introduction

In recent years there has been substantial progress in building machine-checked correctness proofs: for a compiler front-end [8], for a nonoptimizing subset-Pascal compiler [9], and for a multistage optimizing compiler from C to assembly language [10]. These efforts, though they are remarkable and inspiring, do not address the problem of concurrency. Reasoning about concurrent programs, and compiling concurrent shared-memory programs with an optimizing compiler, can be very difficult. The model of computation that programmers might expect does not correspond to what is provided by the machine.

Can we adapt the sequential-language compilers and correctness proofs to the concurrent case by adding threads and locks to their source languages? Not easily. As Boehm explains, "Threads cannot be implemented as a library." [3] An optimizing compiler must be aware of the concurrency model or it might

⋆ Supported in part by NSF Grants 0540914 and 0627650.

⋆⋆ Supported in part by ANR (project ParSec ANR-06-SETI-010-02). January 4, 2008. To appear in European Symposium on Programming (ESOP), April 2008.

inadvertantly break the locking discipline by, for example, changing the order of loads and stores to shared data. Boehm "point[s] out the important issues, and argue[s] that they lie almost exclusively with the compiler and the language specification itself, not with the thread library or its specification." But Boehm does not present a formal semantics: he just explains what can go wrong without one. In this paper we provide the formal semantics that Boehm called for. And we do it in such a way that sequential compilers and proofs preserve their sequential flavor: we will add threads as a kind of *semantic* library.

Contributions. **First** we show that "C + threads" can be specified modularly, by presenting an operational semantics of Extensible C minor. This language is sufficient for compiling C, ML, Java, and other high-level languages. Appel and Blazy [1] have demonstrated a (sequential) Separation Logic, with a machine-checked soundness proof in Coq w.r.t. the small-step operational semantics of *any possible extension* of Extensible C minor.

Second, we present a powerful and expressive Concurrent Separation Logic (CSL) that goes beyond O'Hearn's [11] by permitting dynamic lock and thread creation and by permitting ordinary assertions to describe lock invariants, which are in turn ordinary assertions. Our CSL is very similar to one that Gotsman *et al.* [5] independently developed, demonstrating that it must be *the* natural generalization of O'Hearn's CSL to first-class threads and locks.[3]

Third, we construct the operational semantics of Concurrent C minor, formed by extending Extensible C minor with threads and locks. A novel component of this semantics is a *modal substructural logic* for reasoning about separation in space and evolution in time. Our operational semantics is for well-synchronized programs without data races: any access to a memory location must be performed while holding a lock that gives *ownership* of that location: at least shared ownership for a read and full ownership for a write. Access without ownership causes the operational semantics to get stuck, meaning that the program has no semantics. One can use CSL (using a proof assistant, or via automatic flow analysis [6]) to prove that source programs are well synchronized.

Fourth, from the concurrent operational semantics we will construct a pseudosequential *oracle semantics* for Concurrent C minor. When a sequential thread peforms a concurrent operation such as lock or unlock, the oracle calculates the effect of running all the other threads before resuming back into this thread. We show the correctness of the oracle semantics w.r.t. the concurrent semantics.

The oracle semantics is ideal for reasoning about individual threads—for compilation and flow analysis, and for reusing proofs about the sequential language. Footprint annotations prevent unsound optimizations across lock/unlock operations but are minimally restrictive across sequential operations. The oracle

[3] Our semantic model for CSL is more powerful than Gotsman's in several ways: our model permits assertions to be embedded directly into source code, permits function pointers, recursive assertions, and impredicative quantification; and (unlike Gotsman's) ours connects directly to a small-step sequential operational semantics for a verified-compilable intermediate representation, C minor.

is silent when any of the core sequential control- and data-flow operations are executed, and the operational semantics is deterministic. Therefore, adapting existing machine-checked correctness proofs of the C minor compiler to Oracular C minor should be straightforward.

Fifth, we present a *shallow embedding* of CSL in the Calculus of Inductive Constructions (Coq). A shallow embedding, because it has no induction over CSL syntax, permits new CSL operators to be constructed as needed in a modular way. Our shallow embedding is independent of C-minor statement syntax, thus permitting the insertion of semantic CSL preconditions as annotations in C minor programs.

Finally, we demonstrate that CSL is sound with respect to our oracle semantics, and the oracle semantics is sound w.r.t. the concurrent operational semantics. Thus, properties proved of concurrent C programs will actually hold in machine-language execution.

2 Extensible C minor

Appel and Blazy [1] describe some changes to Leroy's original C minor [10] that make it more suitable for Hoare-Logic reasoning. Expressions can read from the heap but have no side effects. Expression evaluation $\Psi; \sigma \vdash e \Downarrow v$ is with respect to a program Ψ and a *sequential state* $\sigma = (\rho; w; m)$, where ρ is the local-variable environment of the current function activation; and m is the global shared memory. The world w specifies the permissions that this thread has to access memory addresses in m. Worlds enable separation-logic-like reasoning: our semantics gets stuck on loads/stores outside the world. In this presentation we elide many details of C minor; interested readers may consult appendix A.

The sequential small-step relation $\Psi \vdash (\Omega, \sigma, \kappa) \longmapsto (\Omega', \sigma', \kappa')$ operates on *continuations* (Ω, σ, κ) where Ω is an *oracle*, σ is a sequential state, and κ is a control stack:

$$\kappa : \mathsf{control} \ ::= \ \mathsf{Kstop} \mid s \cdot \kappa \mid \ \ldots$$

Kstop is the empty control stack, $s \cdot \kappa$ means "execute the statement s, then continue with κ." C minor has other control operators for function return and nonlocal exit from loops. However, the concurrent semantics is parametric over any syntax of control with at least Kstop and $\cdot$.

Our C minor has a fixed set of control-flow constructs (e.g., if, loop, function call) and straight-line commands (e.g., assign, store, skip). To build an extension, one instantiates *syntax* of additional straight-line commands (e.g. lock, unlock). Then one provides a model of *oracles* to help interpret the additional commands. The oracle contains the state of all the other threads (and the schedule) and calculates what they do when control is yielded. Since our programs are (proved) race-free, preemptive schedules will yield equivalent results. For purely sequential C minor, oracles can be *unit*.

3 Concurrent C minor

We extend C minor with five more statements to make *Concurrent C minor*:

$$s : \mathsf{stmt} \quad ::= \ldots \mid \mathsf{lock}\, e \mid \mathsf{unlock}\, e \mid \mathsf{fork}\, e\, (\vec{e}) \mid \mathsf{make\_lock}\, e\, R \mid \mathsf{free\_lock}\, e$$

The lock (e) statement evaluates e to an address v, then waits until it acquires lock v. The unlock (e) statement releases a lock. A lock at location v is *locked* when the memory contains a 0 at v.

Each lock comes with a *resource invariant* R which is a predicate on world and memory. The invariants serve as a kind of "induction hypothesis" for a correctness or safety proof in CSL, and in particular they tell our operational semantics what addresses are owned by each thread and by each lock, and what addresses are transferred when locking or unlocking. This is standard in CSL [11]; but we go farther and use the invariants at a crucial point in our operational semantics to guarantees the absence of race conditions.

As usual in CSL [11] in order that the resource invariant R will be supported by a unique set of memory addresses in any given memory—these addresses constitute the memory ownership that a thread gains when acquiring a lock or loses when releasing it—the invariant R must be *precise*. The world ($\sim$ set of memory locations) controlled by a lock need not be static; it can change over time depending on the state of memory (one could say, "this lock controls that variable-sized linked list"). When a thread locks a lock, it joins the lock's world with its own; when it later unlocks the lock, it gives up the (possibily different) world satisfying R. This protocol ensures the absence of read/write or write/write race conditions.

The statement make_lock $e\,R$ takes an address e and a lock invariant R, and declares e to be a lock with the associated invariant. The address is turned back into an ordinary location by free_lock e. Both instructions are thread-local (don't synchronize with other threads or any global lock-controller). It is illegal to apply lock or unlock to nonlock addresses, or to apply ordinary load or store to locks.

The fork statement spawns a new thread, which calls function e on arguments $\vec{e}$. No variables are shared between the caller and callee except through the function parameters. The parent passes the child a portion of its world, implicitly specified by the (precise) precondition of the forked function. This portion typically contains visibility (partial ownership) of some locks—then the two threads can communicate. A thread exits by returning from its top-level function call.

We have not added a join operator, since this can be accomplished by the Concurrent C minor programmer by the use of a lock passed from parent to child, unlocked by the child just before exiting.

The concurrent operational semantics checks the truth of lock invariants when unlocking a lock, and checks the truth of function pre- and postconditions when spawning or exiting a thread. Failure of this check causes the operational semantics to get stuck. The language of these conditions contains the full power of logical propositions (Coq's `Prop`), so the operational semantics is nonconstructive: it is given by a classical relation.[4] The lock invariants and the function pre/postconditions can be taken directly from a program proof in concurrent separation logic.

For an example program in Concurrent C minor, see appendix B.1.

4 Concurrent separation logic

We define the usual operators of Separation Logic: **emp**, separating conjunction $*$, disjunction $\vee$, conjunction $\wedge$, and quantifiers $\exists, \forall$. Bornat *et al.* [4] explain the utility of fractional permissions for reasoning statically about alternating concurrent read with exclusive write access, so singleton "maps-to" is extended to support fractional permissions $e_1 \stackrel{\pi}{\mapsto} e_2$. A share can always be split: $e_1 \stackrel{\pi_1}{\mapsto} e_2 * e_1 \stackrel{\pi_2}{\mapsto} e_2 \Leftrightarrow e_1 \stackrel{\pi_1 \oplus \pi_2}{\mapsto} e_2$.

In fact we go beyond fractions, building on the share models presented by Parkinson [12, ch. 5] with a share model of infinite subsets of a countable set (see appendix C). This permits correctness proofs of sophisticated visibility management schemes. But here we will simplify the presentation by just writing 100%, 50%, et cetera. 100% gives permission to read, write, or dispose. Owning $0 < \pi < 100\%$ gives read-only access.

We introduce a new assertion $e \stackrel{\pi}{\bullet\!\!\rightarrow} R$, which means that the expression e evaluates to a memory location containing a lock with resource-invariant R. We write resource(l, R) to mean that R is precise and closed (w.r.t. local variables). A location is either used as a lock or as a mutable reference: a lock assertion $e \stackrel{\pi}{\bullet\!\!\rightarrow} \_$ does not separate from a maps-to assertion $e \stackrel{\pi'}{\mapsto} \_$. Any nonempty ownership π gives the right to (attempt to) lock the lock. An auxiliary assertion, $\mathsf{hold}\ e\, R$, means that lock e with invariant R is locked by "this" thread.

To unlock a lock, the thread must "hold" it: another thread cannot unlock the lock unless the hold has been transferred. Therefore a lock invariant R for lock l must claim the hold of l, in addition to other claims S. That is, $R \Leftrightarrow$

[4] We use a small, consistent set of classical axioms in Coq: extensionality, proposition extensionality, dependent unique choice, relational choice.

$$\frac{\text{resource}(e, R) \qquad R \Leftrightarrow (\mathsf{hold}\ e\, R\ *\ S)}{\Gamma \vdash \{e \stackrel{100\%}{\mapsto} 0\}\mathsf{make\_lock}\, e\, R\{e \stackrel{100\%}{\bullet\!\!\rightarrow} R\ *\ \mathsf{hold}\ e\, R\}}$$

$$\frac{}{\Gamma \vdash \{e \stackrel{100\%}{\bullet\!\!\rightarrow} R\ *\ \mathsf{hold}\ e\, R\}\mathsf{free\_lock}\, e\{e \stackrel{100\%}{\mapsto} 0\}}$$

$$\frac{}{\Gamma \vdash \{e \stackrel{\pi}{\bullet\!\!\rightarrow} R\}\mathsf{lock}\, e\{e \stackrel{\pi}{\bullet\!\!\rightarrow} R\ *\ R\}} \qquad \frac{R \Leftrightarrow (\mathsf{hold}\ e\, R\ *\ S)}{\Gamma \vdash \{R\}\mathsf{unlock}\, e\{\mathbf{emp}\}}$$

$$\frac{\text{precise } (R)}{\Gamma \vdash \{e : \{R\}\{S\}\ *\ R(\vec{e})\}\mathsf{fork}\, e\, \vec{e}\{e : \{R\}\{S\}\}}$$

Fig. 1. Concurrent Separation Logic

$\mathsf{hold}\ l\, R\ *\ S$, where $\Leftrightarrow$ means equivalence of assertions. We achieve this with a recursive assertion $\mu R.(\mathsf{hold}\ l\, R\ *\ S)$, using the μ operator of our CSL.

The assertion that some value f is a function with precondition P and postcondition Q is written $f : \{P\}\{Q\}$. A function can be either called (within this thread) or spawned (as a new thread); but to be spawned, its precondition must be precise: the precondition must specify uniquely the part of the world that the parent passes to the spawned thread.

To handle functions we extend the traditional Hoare triples with an extra context to become $\Gamma \vdash \{P\}s\{Q\}$. The concurrent extension of the logic is independent of the sequential operators and we refer to Appel and Blazy [1] for a description of the sequential logic, in which Γ specifies pre/postconditions of global functions. The concurrent rules are presented in figure 1; appendix B.2 shows our logic applied to an example program.

Impredicativity. Our logic supports both recursive assertions and impredicative polymorphism: one can quantify not only over values and shares, but also assertions. We will use this when describing the lock invariants of object-oriented and higher-order-functional programs, in the same way that impredicative polymorphism is needed in the typed assembly languages of such programs. We also support recursive value-parameterized lock invariants that can describe, for example, "sorted list of lockable cells."

Our CSL does not reason about liveness, and cannot guarantee the absence of memory leaks. Resources can be sent down a black hole by deadlocks, by infinite loops, or by unlocking all of a lock's visibility into its own resource, or by a thread exiting with a nonempty postcondition.

5 A modal model of joinable worlds

Consider the assertion $P = (e \overset{\pi}{\bullet\!\!\to} R)$; here one assertion P describes another assertion R; and maybe R itself describes yet another assertion Q. This makes first-class locks difficult to model semantically. Intuitively, the solution is that P is really a series of increasingly good approximations to the "true" invariant; the kth approximation of P can describe only the $k-1$ approximation of R, which in turn describes only the $k-2$ approximation of S. Then we can do induction on k to reason about the program.

To structure this in a clean way that avoids explicit mention of k, we adapt the "very modal model" of Appel, Melliès, Richards, and Vouillon [2]. They use modal logic to reason about the decrease of k as time advances through the storing and fetching of mutable references. Henceforth we will not mention k explicitly, but it will be implicit in the concept of the *age* of a world.

Our new model advances time as locks are acquired and released. But in addition, now we also reason modally about separation in space. From **machine states** we build a **Kripke model**, which we hide underneath a **modal logic**, which we hide underneath the user view of **Concurrent Separation Logic**.

$P * Q$	separating conjunction
$P \Rightarrow Q \quad P \wedge Q \quad P \vee Q$	implication, (nonseparating) conjuction, disjunction
$\forall v.Q \quad \exists v.Q$	quantification over values, shares, or predicates
$v \overset{\pi}{\mapsto} v'$	v is the address of readable data (writable if $\pi = 100\%$)
$v \overset{\pi}{\bullet\!\!\rightarrow} R$	v is a lock with resource invariant R
$\mathsf{hold}\ v\, R$	the token for "I currently hold the lock v"
$v : \{P\}\{Q\}$	v is a function with precondition P, postcondition Q
μF	recursive: $\mu F = F(\mu F)$
$e \Downarrow v$	the C minor expression e evaluates to v
$[A]_{\mathrm{Coq}}$	formula A in the underlying logic is true
$\mathrm{resource}(l, R)$	R is a valid resource invariant (precise, closed) for lock l
$\mathrm{world}\, w$	the current state's world is equal to w
$\rhd Q$	"later": $Q(\rho, w', m)$ holds in all worlds w' strictly later than w
$\Box Q$	"necessarily": $Q \wedge \rhd Q$
$\bigcirc Q$	"fashionably": $Q(\rho, w', m)$ holds in all worlds w' the same age as w
$!Q$	"everywhere": $Q(\rho', w, m')$ holds on all ρ', m' in the current world
$\mathbf{safe}(\Psi, \kappa)$	with current state σ, for all oracles Ω, stepping $\Psi \vdash (\Omega, \sigma, \kappa) \longmapsto^* \ldots$ cannot get stuck.

Fig. 2. A selection of assertion operators

The Kripke model: $\sigma \Vdash Q$ means that assertion Q holds in a state σ. The forcing relation $\Vdash$ is simple: $Q\sigma$ with Q simply a predicate on states. The world w in $\sigma = (\rho; w; m)$ plays the same role (granting permissions to read/write locations) as did the "footprints" ϕ in Appel & Blazy's Coq proof of sequential-Separation-Logic soundness, which makes it easy to use their proof techniques. The predicates Q of the modal logic are exactly the assertions of the Separation Logic.

Worlds map locations to permissions. Inside the Kripke model (not in the modal logic) we write Val_w^π to describe a nonempty fractional permission π to access a value-cell in world w. The permission $\mathrm{Lock}_w^\pi R$ says that location l is a lock in world w with (nonempty) fractional visibility π. (The subscript w is needed to distinguish the "age" of the Lock permission, as $\mathrm{Lock}_{w'}^\pi R$ in a later world w' has a more approximate semantic meaning.) Fractional visibility of a lock is enough to lock it; 100% visibility (so no other thread can see the lock) is required to deallocate the lock. To model that the locking thread "holds" the lock, and no other thread can unlock it (unless the "hold" is explicitly transferred), we require that R imply (at least) 50% visibility of the lock itself. That is, part of the "visibility" of a lock is really modeling "holding" the lock. The permission $\mathrm{Fun}_w^\pi PQ$ is a function with precondition P and postcondition Q.

Worlds contain lock-permissions; lock-permissions carry assertions; and assertions are predicates on worlds. We resolve this (contravariant) circularity with a stratified construction as shown in Appendix D.

A world describes the *domain* of the heap, where the *contents* of the heap reside in the global memory m. We write $w_1 \oplus w_2$ for the disjoint union of two worlds (where there may be overlap at an address l if the permissions agree and the shares do not exceed 100%). However, $w_1 \oplus w_2$ is only defined if w_1 and w_2 are of the same age; every world in the system ages one tick whenever any thread does a lock, unlock, or fork.

The operators above the line in Fig. 2 are what one might expect in a model of Concurrent Separation Logic. Below the line we have some new modal operators, useful in constructing the semantics but not to be seen by the end user of the Concurrent Separation Logic. *The modalities are contained within our CSL soundness proof.*

Why a modal logic. Suppose we are in world w, and we expect that the current memory m will satisfy predicate Q after *one or more* communications. We write $\rho, w, m \Vdash \triangleright Q$. A lock invariant is an example of a predicate we can only establish "later." To implement higher-order locks, we use the modal logic to keep track of approximations of assertions. We weaken Q every time the clock ticks (i.e., when a thread communicates), and we use $\triangleright$ to keep track of this weakening.

Suppose we lock l that controls world w_l, so our world goes from w to $w' \oplus w'_l$, where primes indicate ticking the clock. By "later" we do *not* refer to the fact that we gain w_l; the modal operator $\triangleright$ talks only about $w \to w'$ or $w_l \to w'_l$. The operator $*$ talks about the $\oplus$ joining. See appendix D for further explanation.

6 Concurrent operational semantics

We specify a concurrent operational semantics to justify the claim that we have a reasonable model of conventional concurrency that corresponds to real machines. The semantics is "world-aware", that is, it gets stuck if a thread attempts to read data for which it has no permission. This means that it must also be "resource-invariant-aware", so that it can transfer the appropriate worlds when locking or unlocking a lock. Therefore, the operational semantics uses the modal logic.

The semantics has two distinct parts. The first part, called the "sequential submachine," executes all instructions that do not depend on other threads, such as call, store, and loop. The second part is fully concurrent; it schedules threads for execution by the sequential part and also handles the explicit synchronization commands: lock, unlock, and fork. Although make_lock and free_lock are new instructions, they do not require synchronization and can be executed by the sequential part of the machine.

This two-part design supports the first half of our modularity principle by hiding the complexities of sequential control- and data-flow from concurrent reasoning. Oracle semantics (section 7) supports the other half by hiding the complexities of concurrent computation from sequential reasoning.

6.1 Sequential submachine

To build the internal sequential submachine, we extend Extensible C minor with the full syntax of all the concurrent instructions and rules for evaluating $\mathsf{make\_lock}$ and $\mathsf{free\_lock}$. The computational result of both of these statements is straightforward, so we use the null oracle $\not{\Omega}$: unit.

To execute $\mathsf{make\_lock}\, e\, R$, the machine evaluates e, ensures that that location is fully owned and currently contains a zero, and updates the world to treat the location as a lock with invariant R. The lock is created with 100% visibility and is held 100% as well.

$$\frac{\begin{array}{cc}\Psi; (\rho; w; m) \vdash e \Downarrow v & \rho, w, m \Vdash (v \overset{100\%}{\mapsto} 0) \;*\; \text{world}\, w_{\text{core}} \\ \rho, w', m \Vdash \text{resource}(v, R) & \rho, w', m \Vdash (v \overset{100\%}{\bullet\!\!\rightarrow} R) * \mathsf{hold}\; v\, R \;*\; \text{world}\, w_{\text{core}}\end{array}}{\Psi \vdash (\not{\Omega}, (\rho; w; m), \mathsf{make\_lock}\, e\, R \cdot \kappa) \;\longmapsto\; (\not{\Omega}, (\rho; w'; m), \kappa)}$$

$\mathsf{free\_lock}\, e$ does the opposite, turning a wholly-owned lock back into a regular location (appendix E.1). At the truly concurrent operations – lock, unlock, fork – the sequential submachine is simply stuck.

6.2 Threads and Concurrent Machine State

The point of a concurrent machine is to execute several threads of control. We define a *thread* θ to be the tuple $(\rho, w, \hat{\kappa})$ with local variables ρ, a private world w, and a *concurrent control-descriptor* $\hat{\kappa}$, defined as follows:

$$\hat{\kappa} : \mathsf{concurrent\ control} \;::=\; \mathsf{Krun}\, \kappa \mid \mathsf{Klock}\, v\, \kappa$$

$\mathsf{Krun}\, \kappa$ means the thread is in a runnable state, with κ as the next sequential control to execute. $\mathsf{Klock}\, v\, \kappa$ means that the thread is waiting on a lock at address v; after acquiring the lock, it will continue with κ. A list of threads we denote by $\vec{\theta}$, and we indicate the ith thread by θ_i.

A *concurrent machine state* $S = (\mho; \vec{\theta}; \mathcal{L}; m)$ has a schedule $\mho$, a (finite) list of thread-ids (natural numbers); a list of threads $\vec{\theta}$; a lock pool $\mathcal{L}$, which is a partial function that associates addresses of *unlocked* locks with the worlds they control; and a memory m. We will be quantifying over all schedules; once given a schedule, C minor executes deterministically, which greatly simplifies reasoning about sequential control-flow [1].

A concurrent machine state also carries with it a set of consistency requirements, ensuring the threads' private worlds are disjoint (among other things; see appendix E.2). In Coq we ensure consistency of concurrent states with a dependently typed record. For this presentation, any concurrent machine state given should be considered consistent.

6.3 Concurrent step relation

The concurrent small-step relation $\Psi \vdash S \Longrightarrow S'$ describes how one concurrent state steps to another in the context of a program Ψ. The full concurrent step relation is given in appendix E.3, but the two critical features are a coroutine interleaving model and a nonconstructive semantics.

Coroutine Interleaving. The concurrent machine context-switches only for fully concurrent operations (lock, unlock, and fork). When executing a series of sequential instructions, the concurrent machine does so without interleaving (thread-number i is not removed from the head of the schedule):

$$\frac{\begin{array}{c}\Psi \vdash (\not\Omega, (\rho; w; m), \kappa) \longmapsto (\not\Omega, (\rho'; w'; m'), \kappa') \\ \vec{\theta}' = [\theta_1, \ldots, \theta_{i-1}, (\rho', w', \mathsf{Krun}\,\kappa'), \theta_{i+1}, \ldots, \theta_n]\end{array}}{\Psi \vdash (i :: \mho; [\theta_1, \ldots, \theta_{i-1}, (\rho, w, \mathsf{Krun}\,\kappa), \theta_{i+1}, \ldots, \theta_n]; \mathcal{L}; m) \Longrightarrow (i :: \mho; \vec{\theta}'; \mathcal{L}; m')}$$

This coroutine model of concurrency may seem strange: it is true that in general it is not equivalent to execution on a real machine. However, our operational semantics permits only well-synchronized programs to execute, so we can reason at the source level in a coroutine semantics and execute in an interleaving semantics or even a weakly consistent memory model. Of course, this claim will require proof: but the proof must be done w.r.t. the machine-language program in a machine-language version of our concurrent operational semantics; this is future work.

Nonconstructive semantics. The noncomputability of our operational semantics arises from the unlock rule:

$$\frac{\begin{array}{ccc}\Psi; (\rho; w; m) \vdash e \Downarrow v & m(v) = 0 & \rho, w, m \Vdash (\mathsf{hold}\ v\ P) * \mathbf{true} \\ w' \oplus w_{\text{lock}} = w & & \boxed{\rho, w_{\text{lock}}, m \Vdash \rhd P} \\ \mathcal{L}' = v : w_{\text{lock}}, \mathcal{L} & \multicolumn{2}{c}{\vec{\theta}' = [\theta_1, \ldots, \theta_{i-1}, (\rho, w', \mathsf{Krun}\,\kappa), \theta_{i+1}, \ldots, \theta_n]} \\ m' = [v \mapsto 1]m & \multicolumn{2}{c}{\text{ContextSwitch } (i :: \mho; \vec{\theta}'; \mathcal{L}'; m') = S}\end{array}}{\Psi \vdash (i :: \mho; [\theta_1, \ldots, \theta_{i-1}, (\rho, w, \mathsf{Krun}\ \mathsf{unlock}\ e \cdot \kappa), \theta_{i+1}, \ldots, \theta_n]; \mathcal{L}; m) \Longrightarrow S}$$

When a lock is unlocked, the semantics checks to make sure that its invariant will hold later ($\rho, w_{\text{lock}}, m \Vdash \rhd P$) – that is, after the unlock operation ticks the clock. If the invariant will not hold, the semantics gets stuck. However, assertions P may contain arbitrary predicates in classical logic—there is no decison procedure for assertions. We are saved by two things: first, if we are executing a program for which we have a proof in CSL, we will know that this check will succeed. Second, if one actually wished to execute a program to see the result, one could execute it on the fully constructive *erased* machine.

An erased machine is simply one that has had all of the worlds and oracles removed, leading to the following much simpler and constructive unlock rule:

$$\frac{\Psi; (\rho, m) \vdash e \Downarrow v \qquad m(v) = 0 \qquad \theta_i = (\rho, \mathsf{Krun}\ \mathsf{unlock}\ e \cdot \kappa) \quad \theta_i' = (\rho, \mathsf{Krun}\,\kappa)}{\Psi \vdash (i :: \mho, [\theta_1, \ldots, \theta_i, \ldots, \theta_n], m) \Longrightarrow (\mho, [\theta_1, \ldots, \theta_i', \ldots, \theta_n], [v \mapsto 1]m)}$$

This is a useful sanity check: the *real* machine takes no decisions based on erasable information; the erased semantics simply approves of fewer executions than the real machine.

When to erase. One could imagine (1) prove safety of a concurrent program w.r.t. the unerased semantics; (2) erase; (3) compile. But this would be a mistake:

$$\text{projection}\ \frac{\Omega = (\mho, \vec{\theta}, \mathcal{L}) \qquad \vec{\theta} = [\theta_1, \ldots, \theta_{i-1}, \theta_{i+1}, \ldots, \theta_n] \qquad \vec{\theta}' = [\theta_1, \ldots, \theta_{i-1}, (\rho, w, \hat{\kappa}), \theta_{i+1}, \ldots, \theta_n]}{(\Omega, (\rho; w; m), \hat{\kappa}) \overset{i}{\propto} (\mho; \vec{\theta}'; \mathcal{L}; m)}$$

$$\text{Ready}\ \frac{\theta_i = (\rho, w, \mathsf{Krun}\, \kappa)}{\text{Ready } i\ (i :: \mho; [\theta_1, \ldots, \theta_n]; \mathcal{L}; m)} \qquad \text{SO-done}\ \frac{\text{Ready } i\ S}{\Psi \vdash \text{StepOthers } i\ S\ S}$$

$$\text{SO-step}\ \frac{\neg(\text{Ready } i\ S) \qquad \Psi \vdash S \Longrightarrow S' \qquad \Psi \vdash \text{StepOthers } i\ S'\ S''}{\Psi \vdash \text{StepOthers } i\ S\ S''}$$

$$\Omega\text{-Invalid}\ \frac{\Omega = (i :: \_, \_, \_) \qquad \nexists S. (\Omega, \sigma, \mathsf{Krun}\, (s_c \cdot \kappa)) \overset{i}{\propto} S}{\Psi \vdash (\Omega, \sigma, s_c \cdot \kappa) \longmapsto (\Omega, \sigma, s_c \cdot \kappa)}$$

Note: s_c ranges over only the concurrent instructions.

$$\Omega\text{-Diverges}\ \frac{\Omega = (i :: \_, \_, \_) \qquad (\Omega, \sigma, \mathsf{Krun}\, (s_c \cdot \kappa)) \overset{i}{\propto} S \qquad \Psi \vdash S \Longrightarrow S' \qquad \nexists S''. \Psi \vdash \text{StepOthers } i\ S'\ S''}{\Psi \vdash (\Omega, \sigma, s_c \cdot \kappa) \longmapsto (\Omega, \sigma, s_c \cdot \kappa)}$$

$$\Omega\text{-Steps}\ \frac{\Omega = (i :: \_, \_, \_) \qquad (\Omega, \sigma, \mathsf{Krun}\, (s_c \cdot \kappa)) \overset{i}{\propto} S \qquad \Psi \vdash S \Longrightarrow S' \qquad \Psi \vdash \text{StepOthers } i\ S'\ S'' \qquad (\Omega', \sigma', \kappa) \overset{i}{\propto} S''}{\Psi \vdash (\Omega, \sigma, s_c \cdot \kappa) \longmapsto (\Omega', \sigma', \kappa)}$$

Fig. 3. Oracle reduction relation $\longmapsto$

as explained by Boehm [3], the compiler may do concurrency-unsafe optimizations. Instead, we must preserve the worlds in the semantics in both source- and machine-language. This gives the compiler a specification of concurrency-safe optimizations. We erase the worlds last, after full compilation.

7 Oracle semantics

A compiler, or a triple $\{P\}c\{Q\}$ in separation logic, considers a single thread at a time. Thus we want a semantics of single-thread computation. The sequential submachine of section 6.1 is single-threaded, but it is incomplete: it gets stuck at concurrent operations. The compiler (and its correctness proof) wants to compile code uniformally even around the concurrent operations. Similarly, in a CSL proof, the commands c_1 and c_2 in $\{P\}c_1;c_2\{Q\}$ may contain concurrent operations, but a soundness proof for the sequence rule of separation logic is complicated enough (because of C minor's nonlocal exits) without adding to it the headaches involved in concurrency. Thus we want a deterministic sequential operational semantics that knows how to handle concurrent communications.

To build the desired semantics, we will build an *oracular machine* using our C minor extension system. As in Section 6.1, we provide the syntax of concurrent C minor. Instead of providing the empty oracle $\not{\Omega}$, however, we define a more meaningful oracle as follows:

$$\Omega : \mathit{oracle} := (\mho, \vec{\theta}, \mathcal{L})$$

An oracle now contains a schedule $\mho$, a list of threads $\vec{\theta}$, and a lock pool $\mathcal{L}$.

We generalize a sequential continuation (Ω, σ, κ) to a concurrent continuation $(\Omega, \sigma, \hat{\kappa})$ whose concurrent control $\hat{\kappa}$ may be ready ($\mathsf{Krun}\,\kappa$) or blocked on a lock ($\mathsf{Klock}\,v\,\kappa$). An oracle allows one to build a concurrent machine S from a thread number i and a concurrent continuation. The precise relationship is given by $(\Omega, \sigma, \hat{\kappa}) \overset{i}{\propto} S$, pronounced "$(\Omega, \sigma, \hat{\kappa})$ is the ith projection of S" (Figure 3).

To execute the extended statements, we use the rules given in Figure 3. For clarity, we use the symbol $\Mapsto$ for the sequential step in oracular C minor, to distinguish from $\longmapsto$ which is the sequential step in the submachine (section 6.1). However, both machines are built with the same C minor extension functors (applied to different oracle types) and therefore have much in common.

When the oracular machine gets to a concurrent instruction, there are several possibilities. The first is that there is no concurrent machine that can be built from the situation given (the rule Ω-Invalid). In this case, the machine loops endlessly, thereby becoming safe. In our proofs we quantify over all oracles—not just valid ones—and this rule allows us to gracefully handle invalid oracles.

In the remaining two cases, we are able to construct a concurrent machine S, and take at least one concurrent step: makelock, freelock, block on a lock (become Klock and context switch), or release a lock (and context switch), or fork (and context switch). After taking this step, the machine decides (classically) if the current thread will ever have control returned to it, by branching on the StepOthers judgement. If the schedule is unfair, if another thread executes an illegal instruction, or if the current thread is deadlocked, then the current thread might never have control returned to it. Rule Ω-Diverges models this by having the machine loop endlessly. The final case is when control returns (rule Ω-Steps); in this case the step proceeds with the new memory, world, and so forth that came from running the concurrent machine.

Classical reasoning in this system is unavoidable: first, the concurrent machine itself requires classical reasoning to find a world satisfying an unlock assertion; second, determining if control will return to a given thread reduces the halting problem. The nonconstructivity of our operational semantics is not a bug: we are not building an interpreter, we are building a specification for correctness proofs of compilers and program logics.

We use the oracular step to keep "unimportant" details of the concurrent machine from interfering with proofs about the sequential language. The key features of the oracular step are: 1) It is deterministic (proof in appendix F.1), 2) When it encounters a synchonization operation, it is able to make progress using the oracle, whereas the regular step relation gets stuck, 3) It composes with itself, whereas the regular step relation does not (because memory will change "between steps" due to other threads), and 4) In the cases where control would never return, such as deadlock, we will be safe.

8 Soundness of CSL on the oracle semantics

In this section we prove that Concurrent Separation Logic is sound with respect to the oracular step. In the next section we prove that the oracular step is sound with respect to the concurrent operational semantics.

A concurrent machine S is *concurrently safe* if, for any S' reachable by $S \Longmapsto^* S'$, either S' can step or its schedule is empty (S' is not stuck). We define $\sigma \Vdash \mathbf{safe}(\Psi, \kappa)$ for a single thread of the oracular machine to mean that $\Psi \vdash (\Omega, \sigma, \kappa) \longmapsto^*$ does not get stuck with any oracle Ω. We call this thread (Ω, σ, κ) *sequentially safe*, written $\Psi \vdash \text{safe}(\Omega, \sigma, \kappa)$. That is, $\mathbf{safe}(\Psi, \kappa)$ is a modal assertion that quantifies over all oracles; $\text{safe}(\Omega, \sigma, \kappa)$ is a predicate on a particular thread with a particular oracle.

Appel and Blazy [1] explain how to model the Hoare tuple $\Gamma \vdash \{P\}c\{Q\}$ in a continuation-passing style. We improve over Appel and Blazy in that our assertions are not predicates over programs. Our global assertion $\Gamma = f_1 : \{P_1\}\{Q_1\} * \cdots * f_n : \{P_n\}\{Q_n\}$ characterizes pre-and post-conditions of global function-names, while theirs characterized function bodies (i.e., syntax). This means that we can embed semantic assertions in program syntax without circularity. However, we are in danger of a different circularity: $\Gamma \vdash \{P\}c\{Q\}$ means "provided that for every $f_i : \{P_i\}\{Q_i\}$ in Γ, the judgment $\Gamma \vdash \{P_i\}\ \Psi(f_i)\ \{Q_i\}$ holds, then command c satisfies its pre- and postcondition," where $\Psi(f_i)$ is the body of function f_i. We solve this problem by defining the Hoare judgment as a recursive assertion. We use the later operator $\triangleright$ to achieve contractiveness, and we tick the clock at function calls. Because of this tick, by the time the caller actually enters a function body, it *will* be later.

$$\begin{aligned}\Gamma \vdash \{P\}c\{Q\} \quad \approx \quad \forall F, \Psi, \kappa.\ (\triangleright \text{ function pre/postconditions in } \Gamma \text{ relate to } \Psi) \Rightarrow \\ F \text{ closed w.r.t. modified vars of } c \ \Rightarrow \\ (\Box \bigcirc!(Q * \Gamma * F \ \Rightarrow\ \mathbf{safe}(\Psi, \kappa))) \ \Rightarrow \\ (\Box \bigcirc!(P * \Gamma * F \ \Rightarrow\ \mathbf{safe}(\Psi, c \cdot \kappa)))\end{aligned}$$

The continuation-passing interpretation of the Hoare triple is, for any frame F, if $Q * F$ is enough to guard κ, then $P * F$ is enough to guard $c \cdot \kappa$. We say $Q * F$ guards κ when any state σ that satisfies $Q * F$ is safe to execute with control κ. Each rule of sequential separation logic is proved as a derived lemma from this definition of the Hoare tuple.

Lemmas: The rules of CSL are proved as derived lemmas from the definition of the Hoare triple. For sequential statement rules, see [1]; for a proof of a concurrent rule, see appendix F.2.

Definition. We write $\Psi \vdash \Gamma$ to mean that for every function mentioned in Γ, its body in Ψ satisfies pre/postconditions of its function declarations. The end-user will prove this using the rules of CSL.

Theorem. Suppose $\Psi \vdash \Gamma$, and $\Gamma \Rightarrow \mathit{main} : \{\mathbf{true}\}\{\mathbf{true}\}$. Then for any n one can construct w_n and a consistent Ω such that $(\Omega, (\rho_0; w_n; m), \mathsf{call}\ \mathit{main}\ () \cdot \mathsf{Kstop})$ is safe to run for at least n communications+function calls.

Corollary. If a program is provable in CSL, then $\mathsf{call}\ main$ is sequentially safe.

9 Concurrent safety from oracular safety

Now we connect the notions of sequential safety and concurrent safety. We say that a concurrent continuation $(\Omega, \sigma, \hat{\kappa})$ is "safe-as i" if, supposing it is the ith thread of the (unique) concurrent machine consistent with its oracle, then if this thread is ever ready and selected then it will be sequentially safe:

$$\frac{(\Omega, \sigma, \hat{\kappa}) \overset{i}{\propto} S \qquad \nexists S'.(\Psi \vdash \text{StepOthers } i\ S\ S')}{\Psi \vdash \text{safe-as } i\ (\Omega, \sigma, \hat{\kappa})} \qquad \frac{(\Omega, \sigma, \hat{\kappa}) \overset{i}{\propto} S \quad \Psi \vdash \text{StepOthers } i\ S\ S' \qquad (\Omega', \sigma', \mathsf{Krun}\,\kappa) \overset{i}{\propto} S' \quad \Psi \vdash \mathsf{safe}\,(\Omega', \sigma', \kappa)}{\Psi \vdash \text{safe-as } i\ (\Omega, \sigma, \hat{\kappa})}$$

Progress. All-threads-safe(S) means that each projection of S will be sequentially safe the next time it is ready and selected; this is enough for progress:

$$\frac{\forall i, \Omega, \sigma, \hat{\kappa}.\ \ (\Omega, \sigma, \hat{\kappa}) \overset{i}{\propto} S\ \rightarrow\ \Psi \vdash \text{safe-as } i\ (\Omega, \sigma, \hat{\kappa})}{\Psi \vdash \text{all-threads-safe}(S)}$$

Lemma. If $\Psi \vdash$ all-threads-safe(S), then S is not stuck. Proof in the appendix F.4.

Preservation. The preservation theorem is more complex due to the existence of forks: we need to know that the child will be safe if its function-precondition is satisfied. To handle this issue, we make the following definition:

$$\frac{\exists \Gamma.\ \forall \rho, w.\ (w \in \vec{\theta}\ \lor\ w \in \mathcal{L}) \rightarrow \qquad \rho, w, m \Vdash (\Psi \vdash \Gamma)\ \land\ (\forall v, P, Q.\ v : \{P\}\{Q\}\ \Rightarrow\ \Box \bigcirc !(\Gamma\ \Rightarrow\ v : \{P\}\{Q\}))}{\Psi \vdash \text{all-funs-spawnable}(\mho, \vec{\theta}, \mathcal{L}, m)}$$

Lemma. If $\Psi \vdash$ all-threads-safe(S), $\Psi \vdash$ all-funs-spawnable(S), and $\Psi \vdash S \Longmapsto S'$, then $\Psi \vdash$ all-threads-safe(S') and $\Psi \vdash$ all-funs-spawnable(S').

Theorem. If each thread is sequentially safe and all functions are spawnable, the concurrent machine is safe.

Corollary. For any schedule $\mho$, if the initial thread $\mathsf{call}\ main\,()$ is sequentially safe and all functions are spawnable, then the concurrent machine is safe.

10 Conclusion

An implementation of C-threads comprises an optimizing C compiler and a threads library implemented in assembly language to handle lock/unlock/fork. From our oracle semantics, we can derive some very simple axioms that the proof of correctness of the optimizing compiler can use. For example, the compiler may wish to hoist loads and stores from one place to another, as dataflow and thread-safety permit. Thread-safety can be captured by simple axioms such as,

$$\frac{\Psi;(\rho;w;m) \vdash e \Downarrow v \qquad w \subset w'}{\Psi;(\rho;w';m) \vdash e \Downarrow v}$$

That is, a bigger world doesn't hurt expression evaluation. To prove $w \subset w'$, we can provide the compiler with rules such as,

$$\frac{\begin{array}{c} c \ = \ \mathsf{loop}\, c' \ \vee \ c = \mathsf{exit}\, n \ \vee \ c = (x{:=}e) \ \vee \ c = \mathsf{if}\, e\, \mathsf{then}\, c_1\, \mathsf{else}\, c_2 \\ \Psi \vdash (\Omega, (\rho;w;m), c \cdot \kappa) \ \longmapsto \ (\Omega', (\rho';w';m'), \kappa') \end{array}}{w = w'}$$

For the extended instructions, the compiler may choose to use no rules at all (so that it cannot hoist loads/stores across calls to functions which may contain lock/unlock), or it may use rules that the world can only grow at a lock or shrink at an unlock. This allows hoisting loads/stores down past lock or up past unlock. All of these rules can be proved sound for our operational semantics.

Our goal in this research has been to provide the compiler with this simple and usable (and proved sound) operational semantics, which in turn is a basis for machine-checked compiler correctness proofs that connect end-to-end (via soundness of CSL) to correctness proofs of concurrent source programs. In future work we hope to connect (at the top) to flow analyses that can produce safety proofs witnessed in CSL, and (at the bottom) to formally prove that machines with weakly consistent memory operations will correctly execute a world-aware machine-level operational semantics that is the output of the compiler. Ideally these should be machine-checked proofs that connect to our Coq proofs of the CSL soundness that we have described here.

All definitions and claims have been fully machine-checked, except that the Coq proofs for Sections 8 and 9 are incomplete; these sections have been proved by hand at the level of rigor traditional for this conference. The concurrent and oracle machines (excluding core C minor) are specified in 1,331 lines; the proofs are 14,430 lines; total *including* sequential C minor and the sequential separation logic soundness proofs is 42,277 lines.

Acknowledgments. We thank Peter O'Hearn and Matthew Parkinson for many interesting and useful discussions.

References

1. A. W. Appel and S. Blazy. Separation logic for small-step C minor. In *20th Int'l Conf. on Theorem Proving in Higher-Order Logics (TPHOLs)*, 2007.
2. A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proc. 34th Annual ACM Symposium on Principles of Programming Languages (POPL'07)*, pp. 109–122, Jan. 2007.
3. H.-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: 2005 ACM SIGPLAN Conf. on Prog. Language Design and Implementation*, pp. 261–268.
4. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL '05*, pp. 259–270, 2005.

5. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *Proceedings 5th Asian Symposium on Programming Languages and Systems (APLAS'07)*, 2007.
6. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI '07: 2007 ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*.
7. A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic (extended version). Tech. report, Princeton University, Jan. 2008.
8. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. on Prog. Lang. and Systems*, 28:619–695, 2006.
9. D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *IEEE Conference on Software Engineering and Formal Methods*, 2005.
10. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL'06*, pp. 42–54, 2006.
11. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1):271–307, May 2007.
12. M. J. Parkinson. *Local Reasoning for Java*. PhD thesis, Univ. of Cambridge, 2005.

APPENDIX

A Simplifications of C minor in our presentation

C minor was designed to be a target language for C, ML, and (sequential) Java compilers. A number of the language features required to do this have been elided from our presentation; although the resulting mass of detail was part of the motivation for the modularity of our model, they make the presentation overly complex. In addition to the ones presented in the body of this paper, major features of C minor include:

- Stacks: each state has a local stack pointer, and there is a global allocation pool for stack frames
- Control-flow: programs have function call, return from the middle of a function body, and multi-level exit from loops. The addition of other control-flow, such as exceptions, would be modularly contained within the sequential portions of our proofs and would not impact the concurrent portions
- Memory model: support for mixed byte- and word-addressibility as well as the ability to distinguish between signed and unsigned data
- Pointer arithmetic: supports ANSI C-style pointer arithmetic

All of these features are handled in our Coq definitions and proofs. Some are quite simple, such as requiring locks to be unsigned 32-bit integers; others are harder, in particular handling the stack space.

B Example program

B.1 Programming with locks.

To illustrate the use of our synchronization operators, we present an example (Figure 4). A thread running main creates a 3-element data block at $\mathsf{L}+1$ con-

```
void main() with pre:{emp} and post:{emp} {
 L := call malloc_and_zero(4);
 i := 0;
 make_lock L R(L);
 fork f(L);
 *(L+3) := 1;
 block {loop {
         if (*(L+3)==0) exit 0 else skip;
         *(L+1) := i; *(L+2) := i;
         unlock L;
         i := i+1;
         lock L;
 }}
 free_lock L;
 call free(L,4);
 return ();
}
```

L	*lock*
L + 1	*data1*
L + 2	*data2*
L + 3	*continue?*

```
void f(l) with pre:{l •—50%→ R(l)} and post:{emp} {
 loop {
  lock l;
  *(l+1) := *(l+1) * 2;
  *(l+2) := *(l+2) * 2;
  if (*(l+1) > 10) {*(l+3) := 0;  unlock l;  return (); }
  else skip;
  unlock l;
}}
```

$$S(l,P) = (\exists v.(l+1 \overset{100\%}{\mapsto} v \; * \; l+2 \overset{100\%}{\mapsto} v) \; * \\ \quad ((l+3 \overset{100\%}{\mapsto} 1) \vee ((l+3 \overset{100\%}{\mapsto} 0) * l \overset{50\%}{\bullet\!\!\to} P))$$

$$R(l) = \mu P.\ (\mathsf{hold}\ l\ P) * S(l,P)$$

Fig. 4. Sample concurrent program

trolled by a lock L with resource invariant $R(\mathsf{L})$ (that will be explained in Section 4). main makes L by converting an ordinary memory location into a lock. It forks a thread running f(L). Further communication is done through the shared data block. When thread f has had enough, it sets word $\mathsf{L}+3$ to 0 and terminates (by returning from f). Then main converts the lock back into an ordinary location of memory, frees its resources, and returns safely.

Lemma (informal). This program is safe.

Proof (informal). The main function cannot deallocate L until after the last time f accesses it; and main cannot deallocate the data block until after the last time f accesses it. This is because main does not exit its loop until it observes $[\mathsf{data}+8] == 0$, which can only happen when f has unlocked L for the last time and is about to return. Moreover, whenever the lock is unlocked, data and $\mathsf{data}+1$ *do* point to the same value: at the beginning, both are zero; after main executes a loop body, both are incremented by one, and if they are the same before f executes a loop body, they will be the same afterwards. Therefore, the final test will succeed, and the program will complete safely.

B.2 Proving the example program safe with CSL

We illustrate some particularities of the logic by proving some statements of the example program in Figure 4. The resource invariant $R(l)$ says that lock l controls three words of data, of which $[l+1] = [l+2]$; and when $[l+3] = 0$ then in addition l controls 50% of its own visibility. Like any lock, it controls its own hold.

Since $R(l) \Leftrightarrow \mathsf{hold}\ l\, R(l) * S(l, R(l))$ is easy via fold/unfold, the $\mathsf{make\_lock}$ in main turns location L into a lock with invariant $R(\mathsf{L})$ (since our logic supports a frame rule, we omit frame contexts):

$$\{\mathsf{L} \overset{100\%}{\mapsto} 0\}\ \mathsf{make\_lock}\ \mathsf{L}\ R(\mathsf{L})\ \{\ \mathsf{L} \overset{100\%}{\bullet\!\!\longrightarrow} R(\mathsf{L})\ *\ \mathsf{hold}\ \mathsf{L}\, R(\mathsf{L})\}\ .$$

The lock invariant is not yet satisfied, but it need not be until the lock is unlocked.

When function f is spawned, main passes to f a partial knowledge of the lock L, as stated in f's precondition:

$$\begin{array}{l} \{\ \mathsf{f} : \{\mathsf{L} \overset{50\%}{\bullet\!\!\longrightarrow} R(\mathsf{L})\}\{\mathbf{emp}\}\ *\ \mathsf{L} \overset{100\%}{\bullet\!\!\longrightarrow} R(\mathsf{L}))\ \} \\ \mathsf{fork}\ \mathsf{f}(\mathsf{L}) \\ \{\ \mathsf{f} : \{\mathsf{L} \overset{50\%}{\bullet\!\!\longrightarrow} R(\mathsf{L})\}\{\mathbf{emp}\}\ *\ \mathsf{L} \overset{50\%}{\bullet\!\!\longrightarrow} R(\mathsf{L}))\ \} \end{array}$$

To $\mathsf{free\_lock}\ \mathsf{L}$ at the end, main must reacquire a full share. It does this just before exiting from the loop. The main loop preserves the invariant $L \overset{50\%}{\bullet\!\!\longrightarrow} R(\mathsf{L})\ *\ R(\mathsf{L})$, and, after taking the if branch, we are left with the loop postcondition

$$((\mathsf{L}+3 \overset{100\%}{\mapsto} 0) * \mathbf{true}) \wedge (L \overset{50\%}{\bullet\!\!\longrightarrow} R(\mathsf{L})\ *\ R(\mathsf{L}))$$

This implies $\mathsf{L} \overset{100\%}{\bullet\!\!\longrightarrow} R(\mathsf{L}) * \mathsf{hold}\ \mathsf{L}\, R(\mathsf{L}) * \ldots$, so $\mathsf{free\_lock}$ is permitted.

In the function f, just before $\mathsf{unlock}\ l;\ \mathsf{return}()$, we dump (50%) ownership of the lock into its own resource invariant! This trick is necessary so that a thread can give up *all* its resources: at the time it unlocks its very last lock, it must still be able to see (have partial ownership of) this lock, but after unlocking it must own nothing. The main thread can then safely dispose the lock L.

C Fractional permissions

A simple, intuitive model of a fractional permission is a rational number in the closed interval $[0, 1]$. However, Parkinson [12, ch. 5] shows that shares represented as subsets of a countably infinite set are more expressive, as one can identify *which half* of the ownership a thread has. (That is, one 50% is not interchangeable with the other; this helps significantly when one needs to define precise predicates, or in reasoning about "token-factory" protocols.)

Parkinson then points out one minor problem with this approach: it is not the case that any nonempty share is splittable into disjoint nonempty shares, as the share might bit a *finite* subset of the universe. He suggests using an isomorphism to solve this problem. We do something much simpler but just as effective: we let the *Share* type be only *infinite* subsets of the universe (or the empty set, representing the empty share).

D Stratified worlds

We want a construction of joinable worlds something like this,

$$\begin{array}{rcl} \mathit{World} & \sim & \mathit{loc} \rightarrow \mathit{Ownership} \\ \mathit{Ownership} & \sim & 0_w + \mathrm{Val}^{\pi}_{w} + \mathrm{Lock}^{\pi}_{w}(\mathit{Assert}) + \mathrm{Fun}^{\pi}_{w}(\mathit{Assert})(\mathit{Assert}) \\ \mathit{Assert} & = & (\mathit{Env} \times \mathit{World} \times \mathit{Mem} \rightarrow \mathrm{Prop}) \\ \mathit{Mem} & = & \mathit{loc} \rightarrow \mathit{value} \end{array}$$

There is a contravariant occurrence of *World* within its own definition that, if understood naively in set theory, would lead to paradoxes. We adapt the "very modal model" of Appel, Melliès, Richards, and Vouillon [2] to avoid this paradox. The intuition is that every resource invariant described in a world w is an approximation, and can be used only in worlds strictly later than w—the clock ticks whenever a communication takes place. The chain of approximations terminates, allowing induction over the accuracy of the approximation. When we say "in world w address l is a lock with resource-invariant R", we mean that "if at w you lock l then you gain resource R and its associated world." The fact that you can only use R in worlds strictly later than w is not a problem, because by the time the lock is obtained the thread is in a later world.

Unlike the previous work, where we are concerned with the semantics of mutable references and we must move to a later world on every machine instruction, in the current work we move to a new world only when a thread performs a concurrent operation (lock, unlock, fork). Leaving the world unchanged at most instructions has the substantial advantage of giving stronger axioms about purely sequential operations (i.e., the after-world is identical to the before-world). Our programs will be safe (or correct) for *any finite schedule*—that is, for arbitrarily long schedules. For any schedule of length n, we start in a world accurate to degree n (i.e., with n successor worlds).

D.1 Construction

We start by defining the theory of joinable things. Type τ is joinable if it has an associative-commutative relation $a \oplus b = c$ and an emptiness test satisfying,

$$\text{empty}(e) \rightarrow e \oplus a = b \rightarrow a = b \qquad \forall a.\exists e.\ \text{empty}(e) \wedge e \oplus a = a$$

$$a \oplus b = c \rightarrow \text{empty}(c) \rightarrow \text{empty}(a) \qquad a \oplus b = b \rightarrow \text{empty}(a)$$

$$a \oplus b = c \rightarrow a \oplus b = c' \rightarrow c = c' \qquad a \oplus b = c \rightarrow a' \oplus b = c \rightarrow a = a'$$

Let joinable(τ) be the type of dependent records containing $\oplus$, empty(), and all these axioms.

Share, described in appendix C, is a joinable type.

A joinable type need not have a unique empty element. Mainly this is because in our construction of worlds, worlds of one age will never be joinable with worlds of a different age, so there must be a different empty world for each age.

Now we want to define *Ownership* as a joinable type, with a join relation such that (e.g.) Val_w^π joins only to $\text{Val}_{w'}^{\pi'}$, and only when π joins with π' and when w, w' have the same age. However, we haven't yet defined worlds and assertions. We define proto-ownerships $\mathit{protoown}(A)$ parameterized by a hypothetical assertion-type A:

```
Inductive protoown(A: Type) : Type :=
  | NO
  | VAL : share -> protoown
  | LK : share ->  A -> protoown
  | FUN: share -> A -> A -> protoown.
```

We omit here the share-nonemptiness proofs carried by `LK`, `VAL`, and `FUN`.

We construct $\mathit{jown}(A) : \text{joinable}(\mathit{protoown}(A))$ by supplying the appropriate $\oplus$ relation and proving the join axioms.

For any joinable type τ, the type $\mathit{loc} \rightarrow \tau$ is a joinable type. Let $w_1, w_2, w : \mathit{loc} \rightarrow \tau$; we say $w_1 \oplus w_2 = w$ iff for every l, $w_1(l) \oplus w_2(l) = w(l)$. For τ a type and $J : \text{joinable}(\tau)$, let joinmap($\tau$)($J$) be the type of dependent records containing J and also containing various extra axioms about joinmaps. For $M : \text{joinmap}(\tau)(J)$, define carrier($M$) = τ.

From $\mathit{jown}(A)$ we construct $\mathit{protoworld}(A) : \text{joinmap}(\mathit{protoown}(A))(\mathit{jown}(A))$, the structure of joinable mappings from address to proto-ownership.

Now we construct "proto-assertions" at level n as,

$$A_0 = \text{unit} \qquad A_{n+1} = (A_n,\ \ \mathit{Env} \times \text{carrier}(\mathit{protoworld}(A_n)) \times \mathit{Mem} \rightarrow \text{Prop})$$

That is, in Coq,

```
Fixpoint protoassert (n: nat) : Type :=
  match n with
   | O => unit
   | S n' => prodT (protoassert n')
            (env -> carrier (protoworld (protoassert n')) -> mem -> Prop)
 end.
```

Now we define protoworld(n) as the joinmap structure on protoown(protoassert(n)), and we define *World* as the dependent pair Σn : Nat.protoworld(n). Let level(w) be the the level of a world (the first component of its dependent pair).

The rest of the model-construction is mostly straightforward, except for the relations between assertions (predicates on worlds) and protoassertions (predicates on protoworlds). For any world w and assertion Q, where level$(w) = n$, we can construct $\lfloor Q \rfloor_w$: protoassert(n) that approximates Q at level n. This is a bit of dependent-type manipulation in Coq that serves to remind us why dependent types should be used only when absolutely necessary in models and proofs.

Now we take $\text{Lock}_w^\pi R$ as $\texttt{LK}(\pi)(\lfloor \Box R \rfloor_w)$.

D.2 Resource Invariants

A lock controls some piece of shared memory. A resource invariant of a lock in CSL is an assertion R that characterizes the heap-portion controlled by the lock, *whenever the lock is in an unlocked state.* One wants R to be a *precise* predicate in separation logic so that the exact portion of memory controlled by the lock can be identified. We also want our lock invariants to be *closed* (that is, to ignore the local variables, since these are not shared between threads) and *extensional* (only depend on those locations in memory where they have the read permission).

We use the standard definition of a *precise* predicate, but we express it within the modal logic, since an assertion might be precise later but not precise now.

$$\begin{array}{l} \rho, w, m \Vdash \text{precise}(Q) \quad \equiv \\ \forall w_1, w_2,\ Q(w_1, m) \wedge Q(w_2, m) \wedge (w_1 \subset w) \wedge (w_2 \subset w) \ \Rightarrow\ w_1 = w_2 \end{array}$$

A predicate is *necessary* if, once it becomes true for some value m, it is true of m in any later world. R is necessary in all worlds if and only if $R = \Box R$. The equality symbol is possible because our assertions are not syntactic: they are semantic definitions on which nontrivial equalities can hold.

What we require of a resource invariant is that it be "later extensional," "later closed," and "later precise," and that it imply the holding of the corresponding lock. We will force it to be necessary by applying $\Box$ to it.

$$\begin{array}{rcl} \text{resource}(l, R) & = & \triangleright \bigcirc!(\text{extensional}(\Box R) \ \wedge \\ & & \quad \text{closed}(\Box R) \ \wedge \\ & & \quad \text{precise}(\Box R) \ \wedge \\ & & \quad (\Box R \Rightarrow \mathsf{hold}\ l\ R * \mathbf{true})) \end{array}$$

If we attempt to require a stronger property, such as "now *and* later precise" instead of just "later precise", then we lose some of the important identities that allow equational reasoning in our semantics (for example, see appendix D.3). However, "later precise" is strong enough to accomplish the following. Suppose lock l has resource R. A thread unlocks l, thus giving up some portion of its world. We need to know exactly what portion to give up. But the clock ticks as the unlock is performed; so instead of splitting worlds before the clock ticks (using the preciseness of R, which may not hold), we could just as well split the world *after* the tick, using the *later* preciseness of R.

D.3 Equivalence of locks

In the modal logic itself we can state the equivalence "from now on" of two predicates, $P \cong Q$: $\Box \bigcirc! \, (P \Rightarrow Q \wedge Q \Rightarrow P)$, by quantifying over the appropriate future worlds and memories. We write $P = Q$ for full extensional equality (over all states past and present).

It would be convenient if whenever $\mathrm{Lock}^{\pi}_{w} R = \mathrm{Lock}^{\pi}_{w} R'$ then $\Box R = \Box R'$, but this is not achievable because of the way $\lfloor \cdot \rfloor_w$ works. The most we can achieve is that $\rhd R = \rhd R'$.

Lemma (Lock Predicate Identity). The following three propositions are equivalent:

$$\rho, w, m \Vdash \ \rhd P \cong \rhd Q$$
$$\mathrm{Lock}^{\pi}_{w} P = \mathrm{Lock}^{\pi}_{w} Q$$
$$\rho, w, m \Vdash \ v \overset{\pi}{\bullet\!\!\longrightarrow} P \cong v \overset{\pi}{\bullet\!\!\longrightarrow} Q$$

That is, from approximate equivalence of predicates P and Q we have full equality of lock specifications. Support for equality allows simpler reasoning in our machine-checked C-minor proofs; we can use substitution tactics in Coq instead of weaker tactics that work on equivalence relations.

E Full Concurrent Machine

E.1 Sequential sub-machine

Figure 5 contains the rules for evaluating make_lock and free_lock.

$$\frac{\Psi;(\rho;w;m)\vdash e\Downarrow v \qquad \rho,w,m\Vdash (v\overset{100\%}{\mapsto}0)\ *\ \text{world}\,w_{\text{core}} \qquad \rho,w',m\Vdash \text{resource}(v,R) \qquad \rho,w',m\Vdash (v\overset{100\%}{\bullet\!\!\rightarrow}R)*\mathsf{hold}\ v\,R\ *\ \text{world}\,w_{\text{core}}}{\Psi\vdash(\mho,(\rho;w;m),\mathsf{make\_lock}\,e\,R\cdot\kappa)\ \longmapsto\ (\mho,(\rho;w';m),\kappa)}$$

$$\frac{\Psi;(\rho;w;m)\vdash e\Downarrow v \qquad \rho,w,m\Vdash (v\overset{100\%}{\bullet\!\!\rightarrow}R)*\mathsf{hold}\ v\,R\ *\ \text{world}\,w_{\text{core}} \qquad \rho,w',m\Vdash (v\overset{100\%}{\mapsto}0)\ *\ \text{world}\,w_{\text{core}}}{\Psi\vdash(\mho,(\rho;w;m),\mathsf{free\_lock}\,e\cdot\kappa)\ \longmapsto\ (\mho,(\rho;w';m),\kappa)}$$

Fig. 5. Sequential steps to reduce pseudo-concurrent operators

E.2 Consistency requirements for concurrent machine states

- The existence of World(S), which is the join of all of the worlds in the threads and all of the worlds in $\mathcal{L}$. This guarantees that all threads have disjoint worlds, implies (in conjunction with precision) the uniqueness of worlds satisfying lock invariants, and insures that all worlds in the machine are the same age.
- World(S) will outlast the schedule.
- For every thread at $\mathsf{Klock}\,v\,\kappa$, v is a lock in the world of that thread.
- For every lock $(l \overset{\pi}{\bullet\!\!\rightarrow} R)$ in World():
  - resource(l, R), that is, l's invariant is a valid resource.
  - $m(l)$ is either 0 (locked) or 1 (unlocked).
  - If $m(l)$ =unlocked, then $\rho, \mathcal{L}(l), m \Vdash \triangleright R$
  - If $m(l)$ =locked, then l is not in the domain of $\mathcal{L}$.

E.3 Concurrent small-step relation

The full concurrent small-step relation is given in figure 6.

The context switch relation, ContextSwitch $(S) = S'$, handles all of the details of performing a context switch by removing the head of the schedule (thus allowing the next thread to execute) and aging all of the worlds.

$$\frac{\vec{\theta}'\text{ is derived from }\vec{\theta}\text{ by aging each thread's world} \qquad \mathcal{L}'\text{ is derived from }\mathcal{L}\text{ by aging each lock's world}}{\text{ContextSwitch }(i::\mho;\vec{\theta};\mathcal{L};m)=(\mho;\vec{\theta}';\mathcal{L}';m)}$$

$$\text{cstep-seq}\ \frac{\begin{array}{c}\Psi \vdash (\not{\Omega}, (\rho; w; m), \kappa) \longmapsto (\not{\Omega}, (\rho'; w'; m'), \kappa') \\ \vec{\theta}' = [\theta_1, \ldots, \theta_{i-1}, (\rho', w', \mathsf{Krun}\,\kappa'), \theta_{i+1}, \ldots, \theta_n]\end{array}}{\Psi \vdash (i :: \mho; [\theta_1, \ldots, \theta_{i-1}, (\rho, w, \mathsf{Krun}\,\kappa), \theta_{i+1}, \ldots, \theta_n]; \mathcal{L}; m) \Longmapsto (i :: \mho; \vec{\theta}'; \mathcal{L}; m')}$$

$$\text{cstep-texit}\ \frac{\text{ContextSwitch } (i :: \mho; [\theta_1, \ldots, \theta_n]; \mathcal{L}; m) = S}{\Psi \vdash (i :: \mho; [\theta_1, \ldots, \theta_{i-1}, (\rho, w, \mathsf{Krun\,Kstop}), \theta_{i+1}, \ldots, \theta_n]; \mathcal{L}; m) \Longmapsto S}$$

$$\text{cstep-prelock}\ \frac{\begin{array}{c}\Psi; (\rho; w; m) \vdash e \Downarrow v \qquad \rho, w, m \Vdash v \overset{\pi}{\bullet\!\!\rightarrow} P * \mathbf{true} \\ \text{ContextSwitch } (i :: \mho; [\theta_1, \ldots, \theta_{i-1}, (\rho, w, \mathsf{Klock}\,v\,\kappa), \theta_{i+1}, \ldots, \theta_n]; \mathcal{L}; m) = S\end{array}}{\Psi \vdash (i :: \mho; [\theta_1, \ldots, \theta_{i-1}, (\rho, w, \mathsf{Krun\,lock}\,e \cdot \kappa), \theta_{i+1}, \ldots, \theta_n]; \mathcal{L}; m) \Longmapsto S}$$

$$\text{cstep-nolock}\ \frac{m(v) = 0 \qquad \text{ContextSwitch } (i :: \mho; [\theta_1, \ldots, \theta_n]; \mathcal{L}; m) = S}{\Psi \vdash (i :: \mho; [\theta_1, \ldots, \theta_{i-1}, (\rho, w, \mathsf{Klock}\,v\,\kappa), \theta_{i+1}, \ldots, \theta_n]; \mathcal{L}; m) \Longmapsto S}$$

$$\text{cstep-lock}\ \frac{\begin{array}{c}m(v) = 1 \qquad m' = [v \mapsto 0]m \qquad \mathcal{L} = v : w_{\text{lock}}, \mathcal{L}' \\ w \oplus w_{\text{lock}} = w' \qquad \vec{\theta}' = [\theta_1, \ldots, \theta_{i-1}, (\rho, w', \mathsf{Krun}\,\kappa), \theta_{i+1}, \ldots, \theta_n]\end{array}}{\Psi \vdash (i :: \mho; [\theta_1, \ldots, \theta_{i-1}, (\rho, w, \mathsf{Klock}\,v\,\kappa), \theta_{i+1}, \ldots, \theta_n]; \mathcal{L}; m) \Longmapsto (i :: \mho; \vec{\theta}'; \mathcal{L}'; m')}$$

$$\text{cstep-unlock}\ \frac{\begin{array}{c}\Psi; (\rho; w; m) \vdash e \Downarrow v \qquad m(v) = 0 \qquad \rho, w, m \Vdash (\mathsf{hold}\ v\ P) * \mathbf{true} \\ w' \oplus w_{\text{lock}} = w \qquad \rho, w_{\text{lock}}, m \Vdash \rhd P \\ m' = [v \mapsto 1]m \qquad \mathcal{L}' = v : w_{\text{lock}}, \mathcal{L} \qquad \vec{\theta}' = [\theta_1, \ldots, \theta_{i-1}, (\rho, w', \mathsf{Krun}\,\kappa), \theta_{i+1}, \ldots, \theta_n] \\ \text{ContextSwitch } (i :: \mho; \vec{\theta}'; \mathcal{L}'; m') = S\end{array}}{\Psi \vdash (i :: \mho; [\theta_1, \ldots, \theta_{i-1}, (\rho, w, \mathsf{Krun\,unlock}\,e \cdot \kappa), \theta_{i+1}, \ldots, \theta_n]; \mathcal{L}; m) \Longmapsto S}$$

$$\text{cstep-fork}\ \frac{\begin{array}{c}\Psi; (\rho; w; m) \vdash e \Downarrow v \qquad \Psi; (\rho; w; m) \vdash \vec{e} \Downarrow \vec{v} \\ \rho, w, m \Vdash (v : \{P\}\{Q\}) * \mathbf{true} \qquad w_{\text{parent}} \oplus w_{\text{child}} = w \qquad \rho, w_{\text{child}}, (\vec{v}, m) \Vdash \rhd P \\ \vec{\theta}' = [\theta_0, \ldots, \theta_{i-1}, (\rho, w_{\text{parent}}, \mathsf{Krun}\,\kappa), \theta_{i+1}, \ldots, \theta_n, (\rho_0, w_{\text{child}}, \mathsf{Krun}\,(\mathsf{call}\,v\,\vec{v} \cdot \mathsf{Kstop}))] \\ \text{ContextSwitch } (i :: \mho; \vec{\theta}'; \mathcal{L}; m) = S\end{array}}{\Psi \vdash (i :: \mho; [\theta_1, \ldots, \theta_{i-1}, (\rho, w, \mathsf{Krun\,fork}\,e\,\vec{e} \cdot \kappa), \theta_{i+1}, \ldots, \theta_n]; \mathcal{L}; m) \Longmapsto S}$$

Fig. 6. The concurrent small-step relation $\Longmapsto$

The rule cstep-seq uses the core semantics of sequential C minor to perform a sequential step, and does not context switch. Thread exit (cstep-texit) does not remove the thread from the list—but scheduling a terminated thread simply results in a context switch. We do not reason about the resources that threads free (or fail to free) on termination.

Three rules describe lock acquisition: cstep-prelock to evaluate the lock address, block (i.e., Klock), and context-switch to give other threads a chance to release the lock; cstep-nolock in case the schedule selects this thread again while the lock is not yet available ($m(v) = 0$); and cstep-lock to actually acquire the lock, moving from Klock to Krun and joining the world w_{lock} that satisfies the lock's resource invariant into the thread's local world.

Finally, cstep-fork creates a new thread. Because we spawn function-calls (not arbitrary commands that might have free C-minor variables), we don't need complex side-conditions on free variables and we can use the empty environment ρ_0 in the new thread. There is a nonconstructive test that the function's precondition is satisfied. The sequential semantics does not need such a test at every function call; we need it here to know what world is transferred by the (precise) predicate P that is the function's precondition. Preconditions of nonspawnable (ordinary) functions need not be precise.

F Various Proofs

F.1 Oracular Determinacy

Lemma. The oracular step $\Vdash\!\!\longrightarrow$ is deterministic.

Proof. Proved in Coq; it follows easily from (1) that the concurrent step is deterministic and (2) the sequential semantics of Extensible C minor is deterministic, given a deterministic extension. Determinacy makes it much easier to do the proofs for sequential control-flow [1] and concurrent CSL rules (appendix F.2).

F.2 CSL lock rule

Lemma. $\Gamma \vdash \{e \overset{\pi}{\bullet\!\!\rightarrow} R\}\mathsf{lock}\ e\{e \overset{\pi}{\bullet\!\!\rightarrow} R\ *\ R\}$.

Proof. Let Ψ be a program, F a frame and κ a control. By the definition of postconditions in the Hoare triple, we assume that for all σ, if $\Psi, \sigma \Vdash (e \overset{\pi}{\bullet\!\!\rightarrow} R) * R * \Gamma * F$ then for all Ω, safe(Ω, σ, κ). Now we are given $\sigma = (\rho; w; m)$ satisfying $\Psi, \sigma \Vdash e \overset{\pi}{\bullet\!\!\rightarrow} R * \Gamma * F$. For an arbitrary oracle Ω, the thread $k = (\Omega, \sigma, \mathsf{lock}\, e \cdot \kappa)$ either reduces to itself (rules Ω-Invalid or Ω-Diverges) and is safe, or takes a step by Ω-steps. Then $k \overset{i}{\propto} S$ and the c-step $\Psi \vdash S \implies S'$ reduces thread i by the rule cstep-prelock. This step, like any that contains ContextSwitch, ages all worlds in S. StepOthers then reduces the concurrent machine to S'', where thread i is rescheduled with control Krun: the last rule applied must be cstep-lock to thread i. Let $k' = (\Omega', \sigma', \kappa) \overset{i}{\propto} S'$, where $\sigma' = (\rho'; w'; m')$. The world w' must be $w \oplus w_{\text{lock}}$ where w_{lock} was associated to the lock e in the lock pool. The consistency requirements of the concurrent machine guarantee that $w_{\text{lock}}, m'' \Vdash \triangleright R$ where m'' is the memory before executing cstep-lock. This guarantees that $\Psi, \sigma' \Vdash (e \overset{\pi}{\bullet\!\!\rightarrow} R) * R * \Gamma * F$. Because of world-aging in at least one cstep, w_{lock} now satisfies R. We deduce safe(k'). Since the oracular semantics is deterministic,[5] states reducing to safe states are safe. We conclude safe(k).

[5] This is one of the cases where a carefully designed deterministic semantics for concurrency is helpful.

F.3 The initial world

Theorem. Suppose $\Psi \vdash \Gamma$, and $\Gamma \Rightarrow main : \{\mathbf{true}\}\{\mathbf{true}\}$. Then for any n one can construct w_n and a consistent Ω such that $(\Omega, (\rho_0; w_n; m), \mathsf{call}\ main\ () \cdot \mathsf{Kstop})$ is safe to run for at least n communications.

Proved in Coq; similar to the proof in Section 12 of [2]. As in that proof the initial world has a simple structure. However, that proof has no oracles. But our initial oracle is also very simple, because the concurrent machine starts with only one active thread.

F.4 Progress

Proof. If the schedule is empty (or picking a nonexistent thread) then S is c-halted. Otherwise $S = (i :: \mho; \vec{\theta}; \mathcal{L}; m)$, and from $\Psi \vdash$ all-threads-safe(S) we can find $\Omega, \sigma, \hat{\kappa}$ such that $\Psi \vdash$ safe-as i $(\Omega, \sigma, \hat{\kappa})$. If $\hat{\kappa} = \mathsf{Krun}\,\kappa$ then $\Psi \vdash$ StepOthers i S S and (by inversion) $\Psi \vdash \mathsf{safe}\,(\Omega, \sigma, \kappa)$. Therefore either $\kappa = \mathsf{Kstop}$ or $(\Omega, \sigma, \kappa) \longmapsto \_$. If $\kappa = \mathsf{Kstop}$ then we can execute cstep-texit. If $(\Omega, \sigma, \kappa) \longmapsto \_$, there are three cases (in Figure 3) which handle concurent statements. Ω-Invalid can be eliminated since θ_i is compatible with S. In both the remaining cases, we have $S \Longmapsto S'$ as a hypothesis. Finally, if it is a core statement of C minor, then we will satisfy the requirements to execute cstep-seq.